Object-oriented programming with Java

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
cc@cse.concordia.ca

August 24, 2018

Disclaimer

- This is a collection of slides for COMP348 (Principles of Programming Languages).
- You are expected to be familiar with programming in Java. These slides present selected topics.

Object creation and initialization

- A class is both a *type* and a *factory*.
- A *constructor* is a special method which automatically initializes an object immediately upon creation.
- A Java constructor has the exact same name as the class in which it resides and it has no return type (not even void).
- During object creation, all *attributes* (also: *data fields*) are set to their default values (integers to *zero*, booleans to *false* and objects to *null*).
- The attributes with initializers are set in the order in which they appear in the class definition. Following that, the constructor body is executed.
- Objects have their own unique set of attributes, but they share methods.

Attributes, parameters and local variables

- The assignment statement `this.author = author;` in the constructor of the class, distinguishes between the attribute `author` (on the LHS of the statement) and the argument with the same name (on the RHS).

```
public class Book {
  private String author;
  ...
  public Book (String author, String title, String year) {
    this.author = author;
    ...
  }
  ...
}
```

Attributes, parameters and local variables /cont.

- This example will not compile since the compiler has no way of distinguishing the local variable `author` (on the LHS of the assignment statement) with the parameter of the same name.

```
public class Book {
  private String author;
  ...
  public Book (String author, String title, String year) {
    String author = author;
    ...
  }
  ...
}
```

Field shadowing

- The statement `String author = "Jill";` defines a local variable which *shadows* the attribute of the same name.
- Hence, the output of the print statement in the constructor is `Constructor: Author is Jill`.
- However, once the body of the constructor terminates, the local variable `author` is discarded and the attribute `author` is `Budd`.

```
private String author;
public Book (String _author, ...) {
  this.author = _author;
  String author = "Jill";
  System.out.println("Constructor: Author is " + author); }
public void displayAuthor () {
  System.out.println ("displayAuthor(): Author is " + author); }

Book b = new Book("Budd", ...);
b.displayAuthor();
```

Type signature

- The *type signature* of a method (or a constructor) is a sequence that consists of the types of its parameters.
- Note that the return type, parameter names, and final designations of parameters are not part of the signature.
- The type signature of the constructor

  `public Book (String author, String title, String year)`

  is $\langle \mathit{String}, \mathit{String}, \mathit{String} \rangle$,
- The type signature of method

  `displayAuthor ()`

  is $\langle \rangle$.

Static features

- *Instance features* (variables and methods) can be accessed only through an object reference.
- *Static features* are used outside of the context of any instances and they may be accessed through either the class name (preferred method) or an object reference.

```
ClassName.staticMethod(parameterList)
ClassName.staticVariable
objectReference.staticMethod(parameterList)
objectReference.staticVariable
```

Example: Static features

```
public class Counter {
  private int value;
  private static int numberOfInstances = 0;
  public Counter() {
    numberOfInstances++;
  }
  public void reset() {
    value = 0;
  }
  public int getValue() {
    return value;
  }
  public void click() {
    value++;
  }
  public static int howMany() {
    return numberOfInstances;
  }
}
```

Example: Static features /cont.

- Unlike instance attributes which can have a different value for each instance of `Counter`, the static attribute `numberOfInstances` is universal to the class.

Example: Static features /cont.

```
public class CounterDemo {
 public static void main (String[] args) {
  Counter c1 = new Counter();
  Counter c2 = new Counter();
  c1.click();
  c1.click();
  c2.click();
  System.out.println("c1 value = " + c1.getValue());
  System.out.println("c2 value = " + c2.getValue());
  System.out.println("Number of Counter objects: "
                     + Counter.howMany());
 }
}
```

```
c1 value = 2
c2 value = 1
Number of Counter objects: 2
```

Example: Static blocks

- *Static blocks* run once as soon as the class is loaded and before the `main()` method executes.

```
public class BlockDemo {
  private static int a = 2;
  private static int b;
  private static void method (int x) {
    System.out.println("Static method: x = " + x);
  }
  static {
    System.out.println("Static block.");
    b = a * 3;
    System.out.println(b);
  }
  public static void main (String[] args) {
    method(13);
  }
}
```

Example: Static blocks /cont.

```
Static block.
6
Static method: x = 13
```

Initialization of static features

1. With their default values as in `private static int numberOfInstances;`
2. With an explicit initializer as in `private static int globalMoveCount = 0;`
3. By a static initialization block:

```
private static int numberOfInstances;
static {
 numberOfInstances = 0;
}
```

Inheritance: Definitions

- We have previously seen that inheritance is a mechanism to define relationships between classes, whereby the definition of one class can extend the definition of another.
- The newly defined class is called a *subclass*[1] and the class whose definition is extended is called a *superclass*.
- A class' immediate superclass is also referred to as its *parent* class.
- All public and protected features of a superclass are accessible in all subclasses.

[1] Some authors use the term *extended*, which is not always true for subclasses.

Inheritance: Definitions

- Inheritance is a mechanism to define relationships between classes.
- Inheritance is transitive, i.e. class can inherit features from superclasses many levels away.
- The collection of classes extending from a common superclass is called an *inheritance hierarchy*.
- The path from a particular class through all its ancestors is called the class' *inheritance chain*.

An initial example on inheritance

- Consider the following inheritance hierarchy:

```
class Person {...}
class Student extends Person {...}
class Professor extends Person {...}
class UndergraduateStudent extends Student {...}
class GraduateStudent extends Student {...}
class TeachingAssistant extends GraduateStudent {...}
```

- If we start from `TeachingAssistant`, the inheritance chain includes: `TeachingAssistant`, `GraduateStudent`, `Student`, `Person`, `Object` (the latter is implicitly included).

Phases of subclass initialization

1. The initialization of the attributes inherited from the superclass (one of the constructors of the superclass must be invoked).
2. The initialization of the attributes declared in the subclass.

Modifiers

- Classes, class features, interface features, method parameters, and local variables can be qualified with *modifiers*.
- A public class is visible to all classes everywhere.
- If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package.
- Class features that are package-private are not accessible to classes defined outside the package, including subclasses of the class.

Modifiers and inheritance

- What happens to inherited features? Based on the types of modifiers attached to features, we can distinguish between the following cases:
- **Public features** can be accessed outside the class definition including outside the package in which they are declared. This is the default modifier for all features declared in an interface.
- **Protected features** can be accessed within the class definition in which they appear, or within other classes in the same package, or within the definitions of subclasses.
- **Private features** can be accessed only within the class definition in which they appear.

Preventing inheritance: Final classes

- We can use the `final` modifier in a class definition to prevent a class from being subclassified.
- We can also make a specific method in a class final in which case no subclass can modify the behavior of this method.

Enforcing inheritance: Abstract classes

- An *abstract class* defines a specification to be inherited. It may also define partial implementation to be inherited.
- An *abstract class* (as opposed to *concrete*) cannot be directly instantiated.
- Any class that contains abstract methods must be declared abstract.
- Any concrete subclass of an abstract class must implement all of the abstract methods defined in the superclass.
- Alternatively, if a subclass implements some (but not all) of the inherited abstract functionality or if it additionally defines its own abstract functionality, then that subclass must itself be declared abstract.

Inheritance and subtyping

- Each class defines a type and all instances of the class constitute legitimate values of that type.
- Every instance of a subclass is also an instance of a superclass, but not vice-versa.
- This implies that the type defined by the subclass is a subset of the type defined by its superclasses as the set of all instances of a subclass is included in the set of all instances of its superclass.

Method overloading

- If two methods or constructors in the same class or in related classes i.e. in a superclass-subclass pair have different type signatures, then they may share the same name.
- We say that they are *overloaded* on the same name with multiple implementations.
- When an overloaded method is called, the number and the types of the arguments are used to determine the signature of the method that will be invoked. Overloading is resolved at compile time.
- In an inheritance hierarchy, we can overload a superclass method to provide additional functionality.

Method overriding

- A subclass can modify behavior inherited from a superclass.
- This is done through a mechanism called *overriding* which refers to the introduction of an instance method in a subclass that has the same name, same type signature and same return type of an inherited method, but a different implementation.
- The implementation of the method in the subclass replaces the implementation of the inherited method from the superclass.

Overriding vs. hiding

- When a subclass declares a variable or static feature that is already declared in a superclass, it is not overridden; it is *hidden*.
- When a hidden feature is invoked or accessed, the copy that will be used is determined at compile time.
- In other words, hidden features are *statically bound*, based on their static type.
- In comparing hiding to overriding, we note that instance methods can only be overridden. When an overridden method is invoked, the implementation that will be executed is chosen at run time.

Static and dynamic type of an object

- Consider an assignment statement of the form

  `type variable = expression;`
- The type that is explicitly mentioned in the assignment statement next to the variable is the variable's *declared* type.
- The variable's declared type is part of its static type: The *static* type of the variable is a set containing the declared type and all its supertypes.
- On the right-hand side of an assignment statement, a variable may be assigned an object of a type different to its static type.
- We call this the *dynamic* (or *run time*) type of the variable.

An initial example on inheritance

```
public class Dog {
 public static void describe() {
  System.out.println("I am a dog."); }
 public void whatIdo() {
  System.out.println("I play in the park."); }}

public class Collie extends Dog {
 public static void describe() {
  System.out.println("I am a dog too!"); }
 public void whatIdo() {
  System.out.println("I save people who are in danger."); }}


Dog lassie = new Collie();
lassie.describe();
lassie.whatIdo();
```

An initial example on inheritance /cont.

- In this example, the declared type of `lassie` is `Dog`, its static type is { `Dog, Object` }, and its dynamic type is `Collie`.
- The `whatIdo()` method in class `Collie` overrides the behavior of the inherited method of the same name and signature from class `Dog`, thus the call to `whatIdo()` will display `I save people who are in danger.`
- However, in the example, variable `lassie` is of type `Dog`, not of type `Collier`.
- The call to the static method `describe()` is resolved statically, i.e. based on the static type of the variable, hence the call to `describe()` will display `I am a dog.`

Subtype relationships

- Classes and interfaces define types and all instances of a class constitute legitimate values of that type.
- The inheritance relationship among classes (and between classes and interfaces) creates a related set of types (*subtype relationship*).
- Type $T1$ is a subtype of type $T2$ if every legitimate value of $T1$ is also a legitimate value of $T2$.
- In this case, $T2$ is the supertype of $T1$. This implies that every instance of a subclass is also an instance of a superclass, but not vice-versa.

Subtype relationships /cont.

- A value of a subtype $T1$ can appear wherever a value of the supertype $T2$ is expected.
- This implies that an instance of a subclass can appear wherever an instance of a superclass is expected.
- The type defined by a subclass is a subset of the type defined by its superclass(-es) as the set of all instances of a subclass is included in the set of all instances of its superclass(-es).

Subtype relationships /cont.

- For example, the pair *(shape, triangle)* defines a subtype relationship as every triangle is a shape (but not vice versa) and thus the set of shapes is a superset of the set of triangles.
- Furthermore, instances of class `Triangle` can appear in any place where an instance of class *Shape* is expected.

Subtype relationships /cont.

- The relationship between *square* and *quadrilateral* in geometry defines a subtype relationship, since a square is a special type of quadrilateral.
- On the other hand, the relationship between *stack* and *vector* does not define a subtype relationship even though it may be practical to deploy inheritance and define the former in terms of the latter.

Compiler responsibilities: Check the validity of assignment statements

- The type of the expression on the right-hand side (RHS) of an assignment statement must the same or a subtype of that of the variable on the left-hand side (LHS).
- In other words, the validity of an assignment statement is based on the static type of a variable.
- This is referred to as *static type checking*.
- In the previous example, the compiler asks "Is `Collie` a subtype of `Dog`?"

Compiler responsibilities:
Check the validity of messages (method calls)

- The compiler needs to verify that the static type of the object (recall that this includes its declared type and all its supertypes) contains a method with a name and signature that can match the message (method call).
- In the previous example the compiler asks "Is there a method `whatIdo()` with signature `()` in the static type of `lassie`?"

Runtime system responsibilities: Choice of method invocation

- A method invocation is determined based on the dynamic type of the object, where the run time system will try to match the message with a method.
- The runtime system will start a lookup from the class definition of the dynamic type of the object.
- If such method exists it will be invoked. Otherwise, the runtime system searches for a match along the inheritance chain until it finds a matching method. This procedure is called *dynamic binding* (or *dynamic dispatch*).
- In the above example, the runtime system will start looking for a method to match `whatIdo()` from the definition of class `Collie`, the dynamic type of `lassie`.

An initial example on inheritance revisited Distinguishing between compiler and run time responsibilities

- `Dog lassie = new Collie();` The assignment statement is valid as `Collie` is a subclass of `Dog`.
- `lassie.describe();` The declared type of `lassie` is `Dog`.
- The static type of `lassie` is { `Dog, Object` }.
- The call to the static method `describe()` is resolved statically, i.e. based on the static type of the variable, hence the call to `describe()` invokes method `describe()` in class `Dog` which will display `I am a dog.`

An initial example on inheritance revisited /cont. Distinguishing between compiler and run time responsibilities

- `lassie.whatIdo();` The compiler needs to check whether the static type of `lassie` contains a method to match the message `whatIdo()`.
- The static type indeed contains such a method and thus compilation is successful.
- The run time system will perform a lookup starting from the dynamic type of `lassie` to locate a method that can match the message.
- Recall that the `whatIdo()` method in class `Collie` overrides the behavior of the inherited method of the same name and signature from class `Dog`.
- The runtime system locates method `whatIdo()` in class `Collie` and invokes it, displaying `I save people who are in danger.`

Interfaces

- An *interface* defines a type, and it is a construct that has evolved over time.
- Up to and include Java 7, an interface encapsulates abstract methods and constants.
- In Java 8, an interface in addition can encapsulate concrete methods (as default) and static methods.
- In Java 9, an interface in addition can encapsulate private methods and private static methods.

Example: Class `Counter`

- In the following example, we have two classes: Class `Counter` and its subclass `LockableCounter`.
- The subclass implements the interface `LockIF`.
- Class `LockableCounter` inherits all methods from `Counter` and implements all methods declared in `LockIF`.

Example: Class Counter /cont.

```
public class Counter {
  static String description = "The foundation of all counters.";
  int value;
  public void reset() {
    value = 0;
  }
  public int getValue() {
    return value;
  }
  public void click() {
    value++; }}


public interface LockIF {
  void lock();
  void unlock();
  boolean isLocked(); }
```

Example: Class Counter /cont.

```
public class LockableCounter extends Counter implements LockIF {
  static String description = "A lockable counter.";
  private boolean lock;
  public void lock() {
    this.lock = true;
  }
  public void unlock() {
    this.lock = false;
  }
  public boolean isLocked() {
    return this.lock;
  }
  public void click() {
    this.value = this.value + 2;
  }
}
```

Example: Class Counter /cont.

```
Counter c1 = new LockableCounter();
LockableCounter c2 = new LockableCounter();
LockIF c3 = new LockableCounter();
c1.click();
System.out.println(c1.getValue());
System.out.println(c1.description);
c2.click();
System.out.println(c2.getValue());
c2.unlock();
System.out.println(c2.isLocked());
System.out.println(c2.description);
//c3.click();
c3.unlock();
System.out.println(c3.isLocked());
//System.out.println(c3.description);
```

Example: Class `Counter` /cont.

- The statement

  ```
  Counter c1 = new LockableCounter();
  ```

  will be checked (statically) by the compiler. Class `LockableCounter` is a subclass and therefore a subtype of class `Counter`.
- Similarly, the statements

  ```
  LockableCounter c2 = new LockableCounter();
  LockIF c3 = new LockableCounter();
  ```

  are valid and type checking will be successful.

Example: Class `Counter` /cont.

- For the statement `c1.click()` the compiler will check to see if a matching method to message `click()` is defined in the static type of the object, namely `Counter`. As a result, method invocation is valid.
- The runtime system will make a decision which method to call based on the dynamic type of the variable.
- The dynamic type of the variable is `LockableCounter` and the runtime system will invoke the overriding method `click()`.

Example: Class `Counter` /cont.

- The statement `System.out.println(c1.getValue())` is statically valid because method `getValue()` is defined in the static type of the variable, namely class `Counter`.
- The statement will display 2 because the overriding method `click()` has incremented the variable `value` by 2.

Example: Class `Counter` /cont.

- The statement `System.out.println(c1.description)` accesses a static variable.
- The binding is based on the static type and this will display "The foundation of all counters."
- The statement `c2.click()` will be successfully statically checked as there is a method match to the message `click()` in the static type of the variable, namely `LockableCounter`.
- The runtime will call the method of the dynamic type, namely the one in `LockableCounter`.

Example: Class `Counter` /cont.

- The statement `System.out.println(c2.getValue())` will be successfully statically checked and display 2.
- The statement `c2.unlock()` will be successfully statically checked as there exists a matched method to `unlock()` in the static type of the variable, namely the interface `LockIF`.

Example: Class `Counter` /cont.

- The statement `System.out.println(c2.isLocked())` will display *false*.
- The statement `System.out.println(c2.description)` is an example of hiding.
- As the binding of a static variable is done on the static type of the variable, the statement accesses the value of `description` variable in class `LockableCounter`, thus displaying "A lockable counter."

Example: Class `Counter` /cont.

- The statement `c3.click()` will fail type checking because there is no matching method in the static type of the variable `c3`, namely the interface `LockIF`.
- The statement `c3.unlock()` will be successfully statically checked as there is a matching method to the message `unlock()` in the static type of the variable `c3`.
- The statement `System.out.println(c3.isLocked())` will be successfully statically checked and it will display *false*.
- The statement `System.out.println(c3.description)` will fail.

Resolving name collisions of interface features

- To avoid problems associated with multiple inheritance, Java allows only single inheritance for class extension but allows multiple inheritance for interface implementation.
- This does not guarantee that no potential problems may show up, as name collisions may exist between features of different interfaces or between features of interfaces and classes.

Casting

- We can convert between types as follows: the conversion of a subtype to one of its supertypes is called *widening* and it is carried out implicitly whenever necessary.
- In other words, a reference to an object of class `C` can be implicitly converted to a reference to an object of one of the superclasses of `C`.
- On the other hand, the conversion of a supertype to one of its subtypes is called *narrowing*.
- Narrowing of reference types requires explicit casts.

Narrowing: Responsibilities of the compiler

- Consider class `Parent`, being the superclass of `Child` and object `p` being an instance of `Parent`. Let `Child` define method `calculate()`.
- In `((Child)p).calculate();` both the compiler and the runtime system are involved in validating the explicit casting as follows:
- The compiler will obtain the static type of `p`, namely `Parent`. Next, the compiler needs to ensure that the object is (in the case of downcasting) casted downwards in its inheritance chain which is indeed the case in this example. In other words, one cannot cast an object to a non-related type.

Narrowing: Responsibilities of the compiler /cont.

- If, in our example, we had class `Friend` that contained method `calculate()` and we attempted to do `((Friend)p).calculate();` the compiler will issue an error.
- Back to our initial example, the compiler will proceed to check if a method signature exists that matches the message `calculate()` in the static type of the object, that includes the now "forced" type, namely `Child` and all its supertypes. Compilation is successful.

Narrowing: Responsibilities of the runtime system

- The runtime system must ensure that the dynamic type of object `p` is `Child` (or one of its subtypes).
- Once this validation is successful, the runtime system will invoke method `calculate()` in the class that corresponds to the dynamic type, in this case class `Child`.
- If not successful, the runtime system will throw an exception.

Motivation for narrowing

- What is the motivation for narrowing?
- Recall that the functionality of a superclass is available to all subclasses and that the subclasses normally contain extended functionality.
- Narrowing allows us to temporarily achieve the opposite: to extend the functionality of a superclass with that of a subclass.

An initial example on casting

```
public class Parent { }
public class Child extends Parent {
  public String greet () {return "Good morning!";}}
public class Grandchild extends Child {
  public String greet () {return "I want ice cream.";}}
public class Friend {
  public String greet () {return "Hello there!";}}

Parent p = new Child();
System.out.println(((Friend)p).greet());
```

Compilation: Not successful.
Runtime and output: N/A.

An initial example on casting

```
public class Parent { }
public class Child extends Parent {
  public String greet () {return "Good morning!";}}
public class Grandchild extends Child {
  public String greet () {return "I want ice cream.";}}
public class Friend {
  public String greet () {return "Hello there!";}}

Parent p = new Child();
System.out.println(((Child)p).greet());
```

Compilation: Successful.
Runtime and output: Good morning!

An initial example on casting /cont.

```
public class Parent { }
public class Child extends Parent {
  public String greet () {return "Good morning!";}}
public class Grandchild extends Child {
  public String greet () {return "I want ice cream.";}}
public class Friend {
  public String greet () {return "Hello there!";}}

Parent p = new Parent();
System.out.println(((Child)p).greet());
```

Compilation: Successful.
Runtime and output: Exception thrown: Parent cannot be cast to Child.

An initial example on casting /cont.

```
public class Parent { }
public class Child extends Parent {
  public String greet () {return "Good morning!";}}
public class Grandchild extends Child {
  public String greet () {return "I want ice cream.";}}
public class Friend {
  public String greet () {return "Hello there!";}}

Grandchild p = new Child();
System.out.println(((Child)p).greet());
```

Compilation: Not successful.
Runtime and output: N/A.

An initial example on casting /cont.

```
public class Parent { }
public class Child extends Parent {
  public String greet () {return "Good morning!";}}
public class Grandchild extends Child {
  public String greet () {return "I want ice cream.";}}
public class Friend {
  public String greet () {return "Hello there!";}}

Parent p = new Grandchild();
System.out.println(((Child)p).greet());
```

Compilation: Successful.
Runtime and output: I want ice cream.

An initial example on casting /cont.

```
public class Parent { }
public class Child extends Parent {
  public String greet () {return "Good morning!";}}
public class Grandchild extends Child {
  public String greet () {return "I want ice cream.";}}
public class Friend {
  public String greet () {return "Hello there!";}}

Parent p = new Child();
System.out.println(((Grandchild)p).greet());
```

Compilation: Successful.
Runtime and output: Exception thrown: Child cannot be cast to Grandchild.