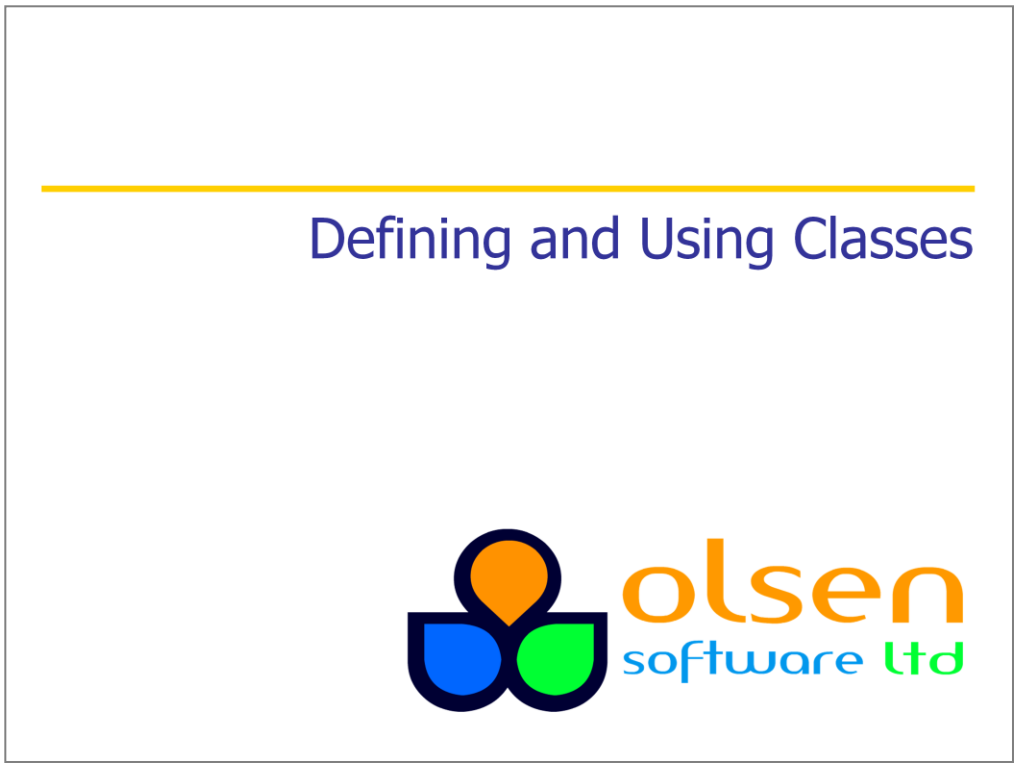


In this chapter we'll start looking at how to write object-oriented Java code. We'll describe how to define classes, and then see how to create, initialize, and use objects. We'll also discuss some important related issues such as `static` members.

Contents

Demo project: **DemoClasses**

2

Section 1 introduces essential OO concepts - e.g. what is a class, what is an object, what is encapsulation, etc.

Section 2 describes the basic syntax for defining classes. As a Java developer, you spend all your time defining classes, so this is very important.

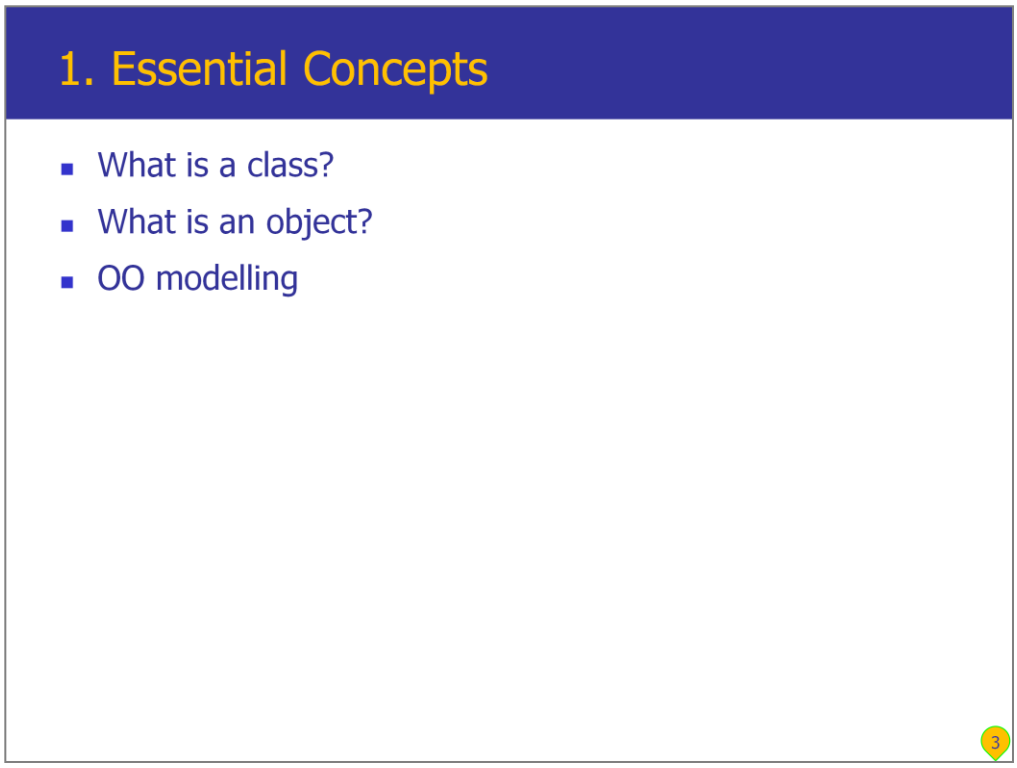
Section 3 delves into objects. We show how to create a new object, and how to use the object in the rest of the program.

Section 4 takes a closer look at object initialization. It's all about constructors…

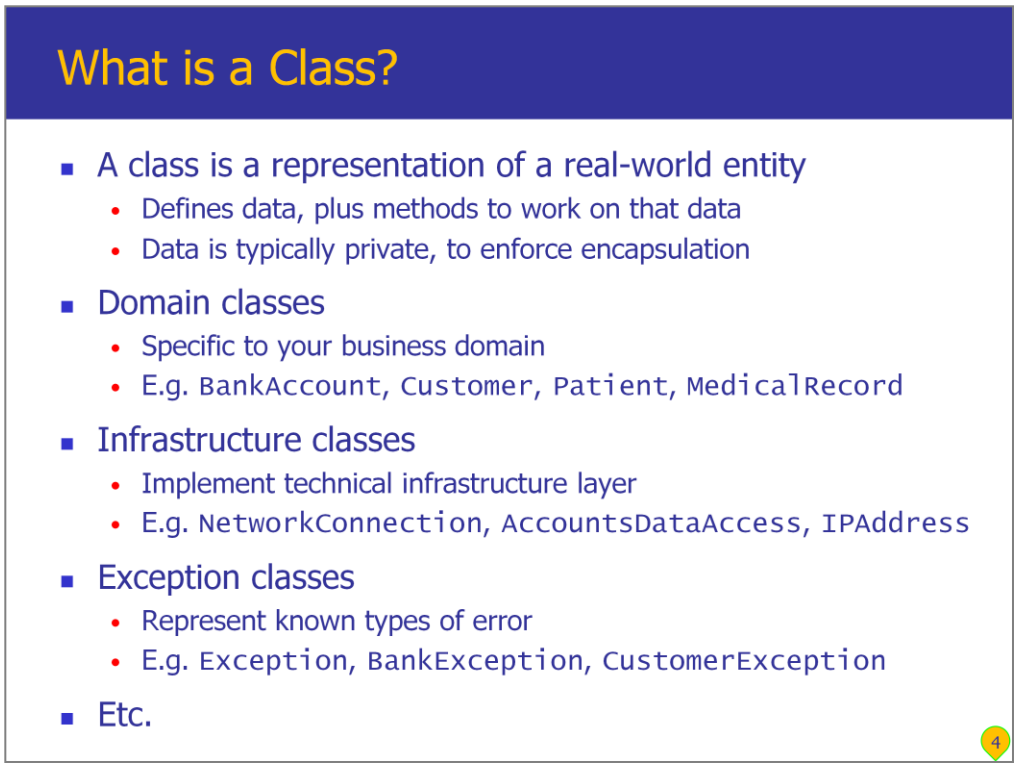
Section 5 describes the `static` keyword. We've seen `static` a few times now, so this is where we get to give a formal definition.

There are also two optional "Annex" sections at the end of the chapter, which describes UML diagrams and JavaBean events respectively. We won't cover these Annex sections explicitly during the lecture, but feel free to peruse the slides if you get a moment.

The demos for chapter are located in the `DemoClasses` project.



Before we dive into the details of how to define classes and objects in Java, we'll spend a moment defining what these terms are, and outlining the purpose and benefits of object-oriented modelling.



Technically, a class is a data type that allows you to define related data and operations in one place. It's a bit like a data structure, but you typically define the data members as private and encapsulate them with a set of public operations. From a terminology perspective, we use the term "method" rather than "operation" in Java.

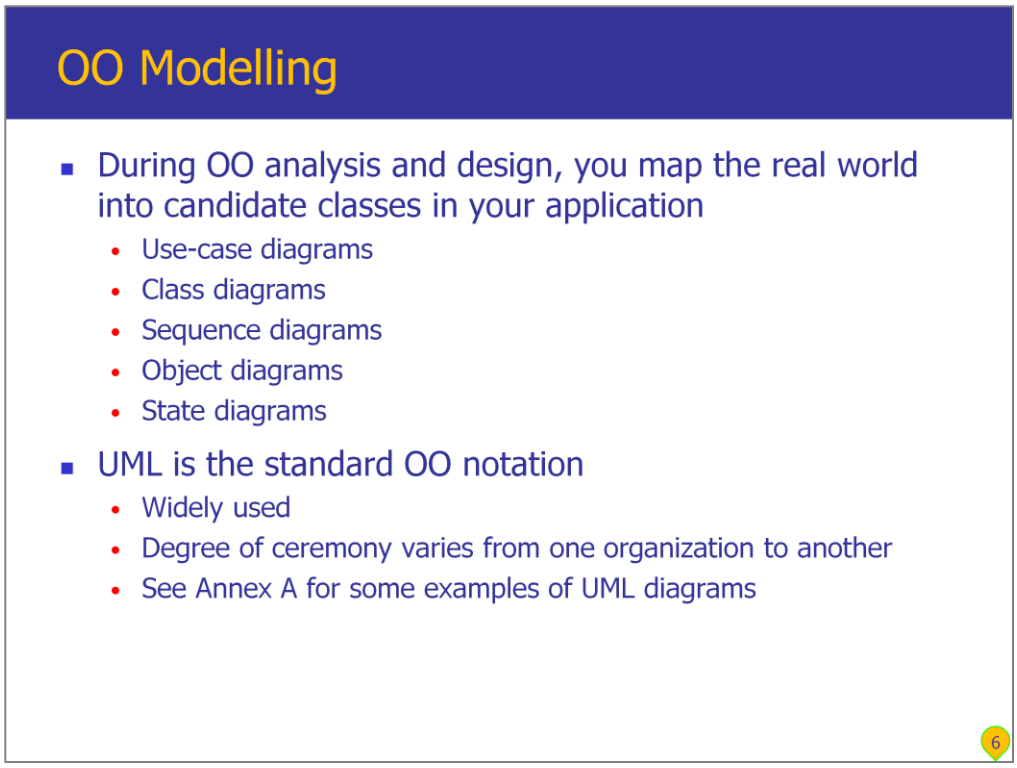
The Java SE library contains thousands of predefined classes, which you will become familiar with on an as-needed basis during your Java programming career. You will also define your own classes, to represent the kinds of entity relevant to your domain.

What is an Object?

- An object is an instance of a class
  - Created (or "instantiated") via the new operator
  - Each object is uniquely referenced by its memory address (no need for primary keys, as in a database)
- Object management
  - Objects are allocated on the garbage-collected heap
  - An object remains allocated until the last remaining object reference disappears
  - At this point, the object is available for garbage collection
  - The garbage collector will reclaim its memory sometime thereafter

5

An object is an instance of a class. You create an object by using the new operator. The object lives on the garbage-collected heap until it is no longer referenced anywhere in your application. When the object is no longer referenced, it becomes available for garbage collection. The garbage collector runs periodically, in a non-deterministic kind of way, to free up the memory used by unreferenced objects.



The Unified Modelling Language is a standard and well-respected notation for expressing object oriented analysis and design decisions. There are several types of diagram you can create:

- Use-case diagrams are useful during analysis. You can identify what kinds of users will use the system, and what functionality they need from the system.
- Class diagrams capture information about what classes will exist in the system, and how they relate to each other.
- Sequence diagrams show how objects will interact with each other dynamically when the application is running.
- Object diagrams are similar to class diagrams, except they show object instances rather than classes. For example, an object diagram might show two `DatabaseManager` objects and describe how they represent a primary data store and a secondary data store in the application.
- State diagrams are useful if a class's behaviour is highly dependent on its current state. A state diagram models the different states an object can be in, and identifies the operations allowed on an object in each of these states.

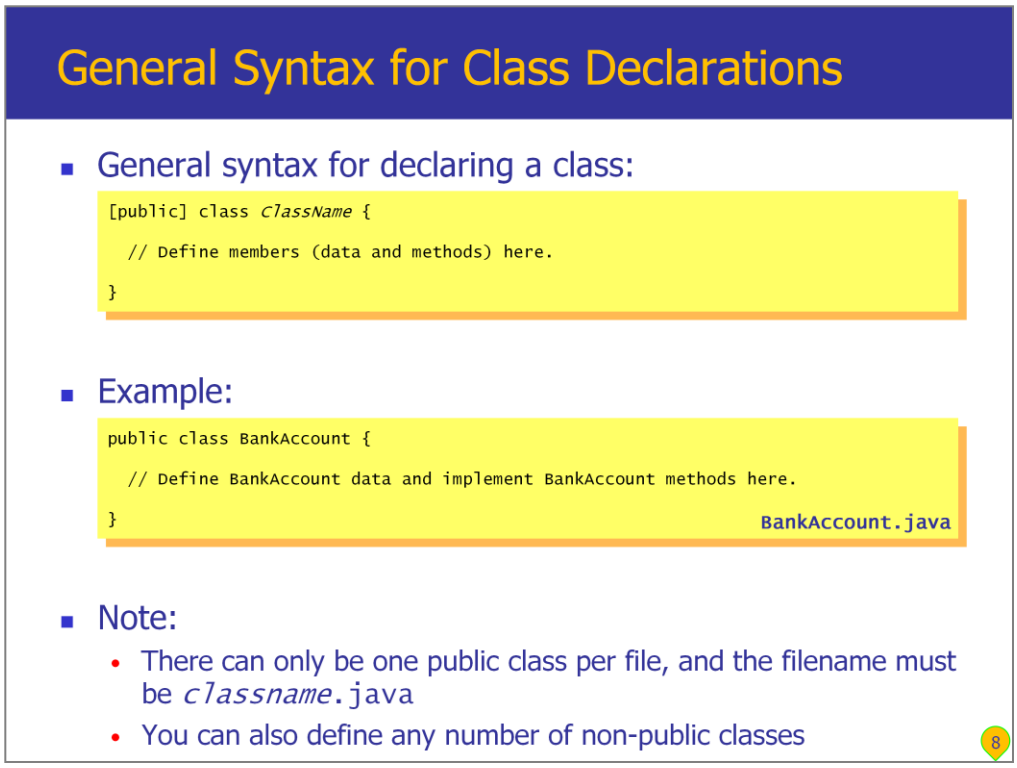
For more information about UML, see http://www.uml.org/

2. Defining a Class

- General syntax for class declarations
- Member modifiers
- Defining instance variables
- Defining getters and setters
- The this keyword
- Defining instance methods
- Overloading methods
- Overriding methods

7

It's time to look at some syntax! This section shows how to define classes in Java, including data members and methods. We'll also discuss related topics such as method overloading and method overriding.



The first code box in the slide shows the general syntax for defining a class. The `public` keyword is optional. If you omit the `public` keyword, the class will only be visible to other classes defined in the same package.

The second code box shows an example. It shows how to define a public class named `BankAccount`. According to the basic rules of Java, you must place this code in a file named `BankAccount.java`.

Note that you should also define a package for all your classes. For example:

```
public mypackage;

public class BankAccount {
   …
}
```

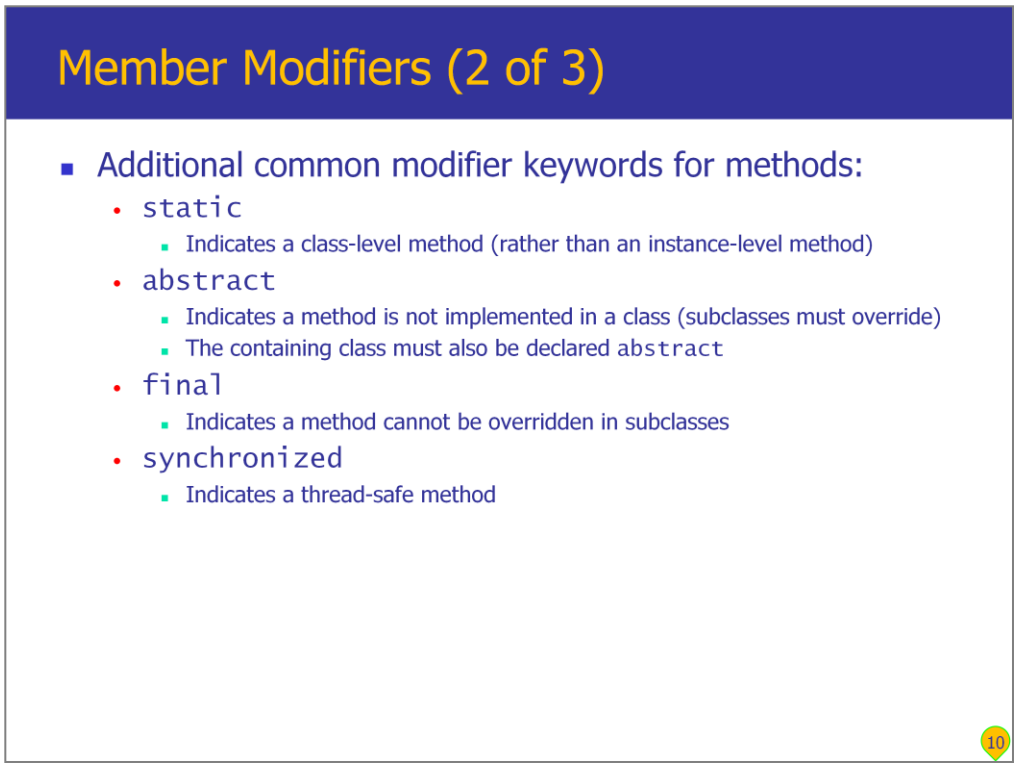
Member Modifiers (1 of 3)

- Java supports 4 access levels for members in a class…
  - `public`
    - Accessible by anyone
    - Methods and constants are often `public`
  - `private`
    - Accessible only by class itself
    - Data and helper methods are usually `private`
  - `protected`
    - Accessible by class itself, subclasses, and classes in same package
    - Allow access to members that are hidden from general client code
  - (No access modifier)
    - Accessible by class itself, and classes in same package

9

Within a class, you will define various members (i.e. data members and methods). Each member will have its own access level, as described in the slide.

Generally you should define as many members as possible as `private`. This maximizes the encapsulation of the class - i.e. it makes the class internals hidden from all other classes. This makes it easier for you to change how the class is implemented internally, without breaking any external code. Encapsulation is an extremely important tenet of OO programming.

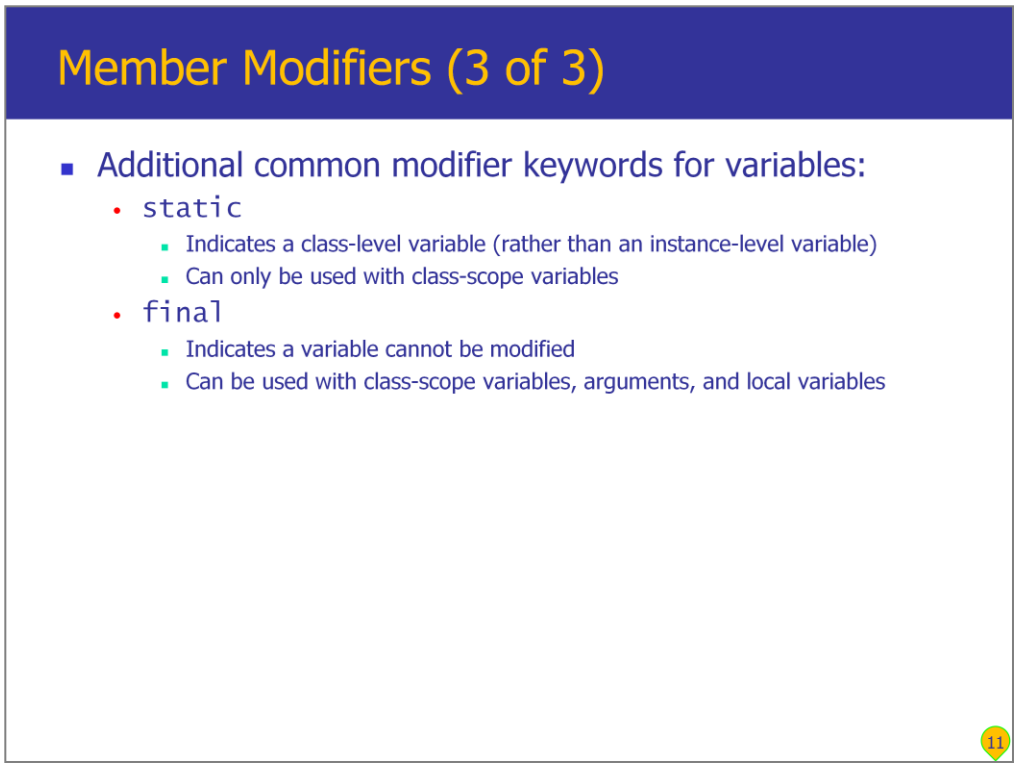


This slide describes some additional keywords that you can use when you define a method in a class. Note the following points:

- The `static` keyword specifies a class method rather than an instance method. We'll discuss `static` later in this chapter.
- The `abstract` and `final` keywords pertain to inheritance, so we'll discuss these in detail in the Inheritance chapter.
- The `synchronized` keyword pertains to thread safety, so we'll discuss it in detail in the Multithreading chapter.

Note that Java also supports two additional (and advanced) modifiers for methods, not shown on the slide:

- `native`
  - Indicates a method is implemented in native code (usually C), and is not implemented in the Java class definition. For more information, see http://en.wikibooks.org/wiki/Java_Programming/Keywords/native.
- `strictfp`
  - Indicates a method uses IEEE 754 floating point maths. For info, see http://en.wikibooks.org/wiki/Java_Programming/Keywords/strictfp.

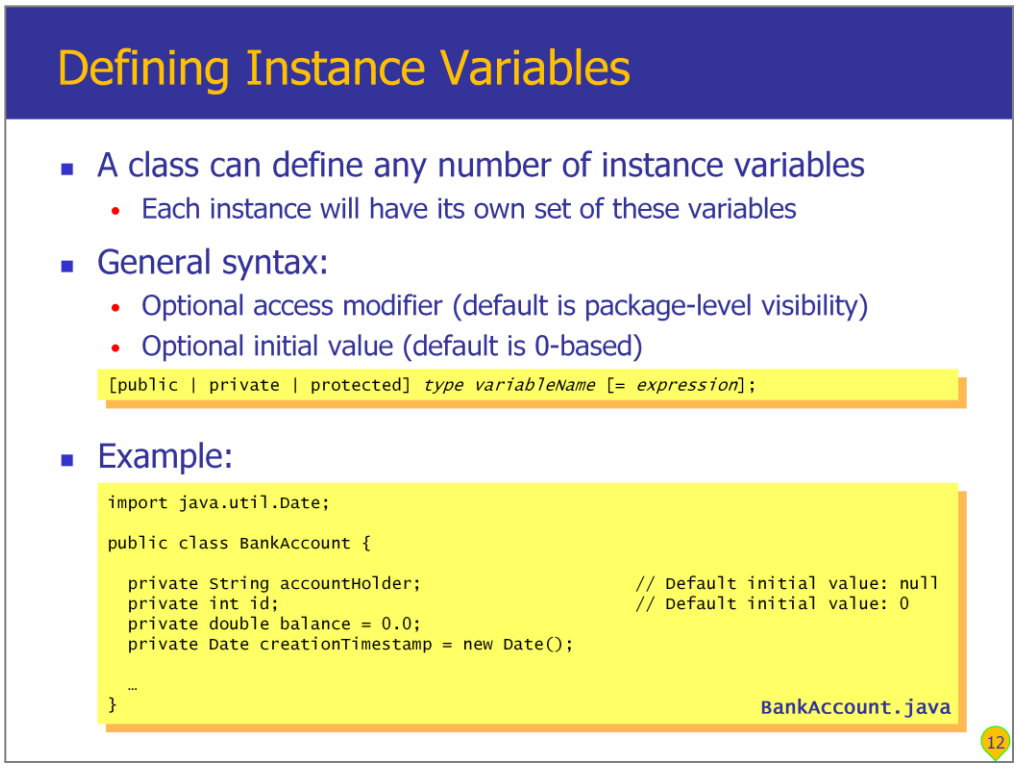


This slide describes some additional keywords that you can use when you define a data member in a class. Note the following points:

- The `static` keyword specifies a class data member rather than an instance data member. We'll discuss `static` later in this chapter.
- The `final` keyword specifies a data member that cannot be modified after its declaration. In other words, it means the variable is a constant.

Note that Java also supports two additional (and advanced) modifiers for class-scope variables, not shown on the slide:

- `volatile`
  - Indicates a variable might be modified out-of-band (i.e. by a different thread). We'll discuss this in more detail in the Multithreading chapter.
- `transient`
  - Indicates a variable should not be serialized. We'll discuss serialization in the File Handling chapter.

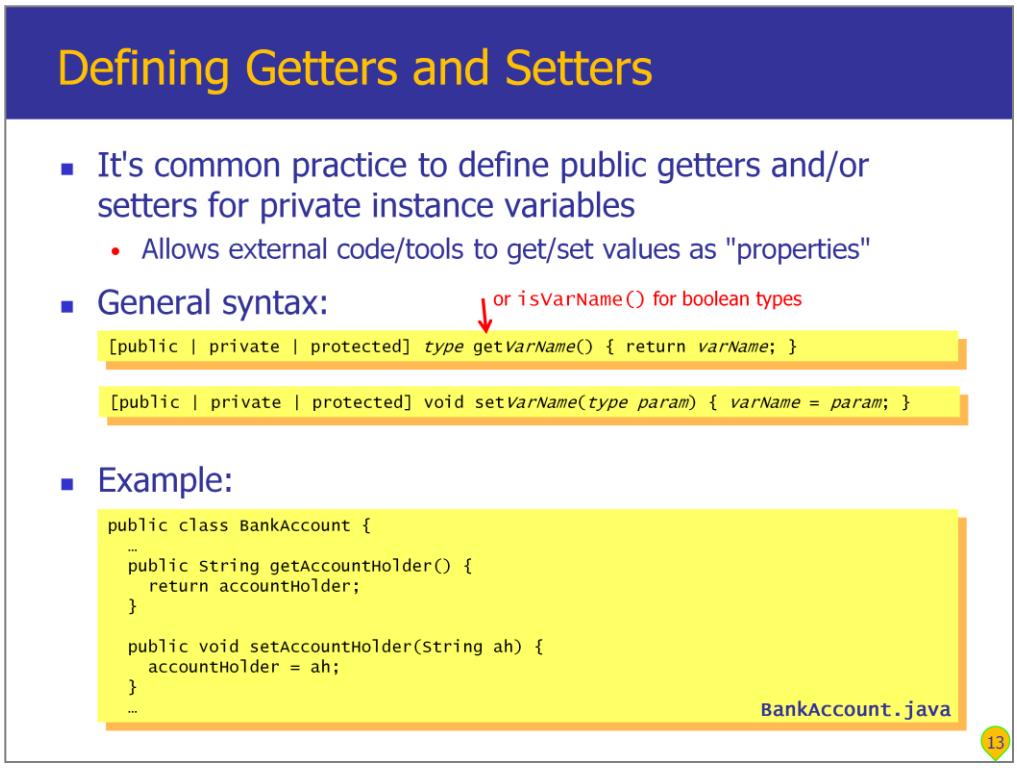


Now that we've got some of the theory out of the way, it's time for some examples. The code in the slide shows how to declare some data members in the `BankAccount` class. Note that all the data members are `private`, which is recommended best practice.

Note the following points about the instance variables in the `BankAccount` class:

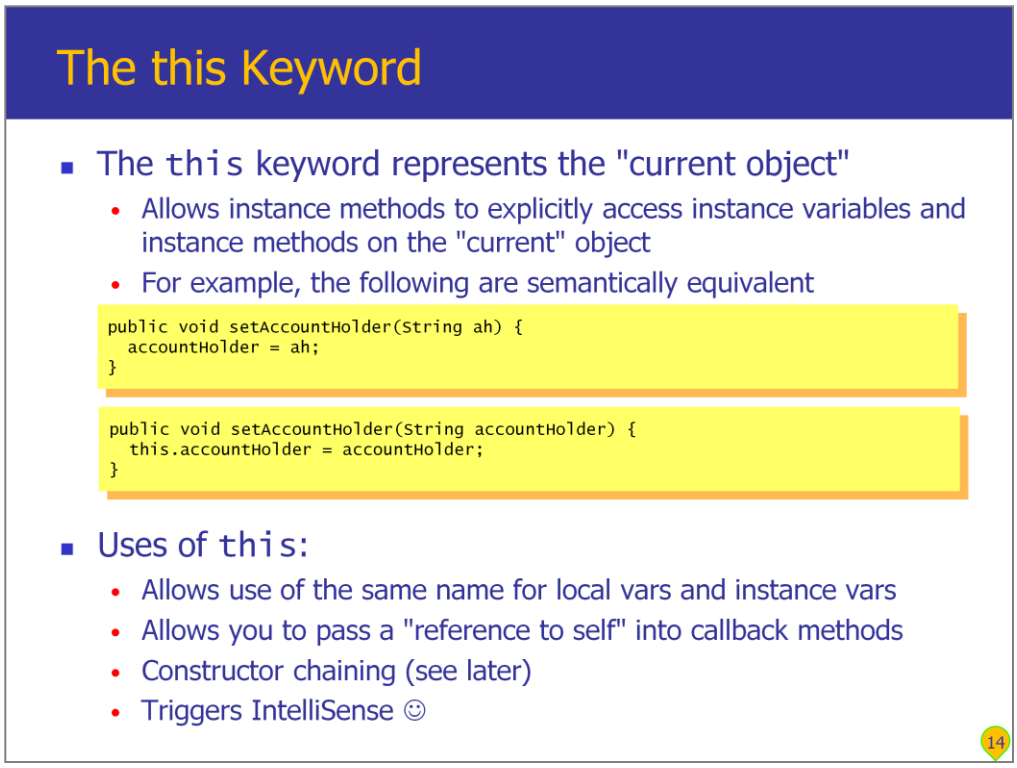
- `accountHolder` is a `String` instance variable. It isn't initialized explicitly, so it assumes a default initial value of `null`.
- `id` is an `int` instance variable. It isn't initialized explicitly either, so it assumes a default initial value of `0`.
- `balance` is a `double` instance variable, and is explicitly initialized to `0.0`. This would have been the default initial value anyway, so this explicit initialization is redundant here.
- `creationTimestamp` is an `Date` instance variable, and is explicitly initialized to refer to a new `Date` object. When you create a `Date` object in this way, it contains the current date and time. Note that the `Date` class is defined in the `java.util` package, hence the `import` statement at the top of the code sample.

The full code for the `BankAccount` class is available in the demo project.



As noted on the previous slide, it's recommended best practice to define data members as `private`. This prevents external code from accessing the data directly, which improves the encapsulation of your class.

If you really do need external code to get or set the value of a data member, you can define getter and/or setter methods. There's a strict naming convention for these methods, as shown in the slide. This naming convention allows tools (such as IDEs) to provide design-time access to the "properties" in your class. Also see Annex B at the end of this chapter, for event naming convention.



When you implement an instance method (i.e. a method that isn't qualified with the `static` keyword), the method has an implicit variable named `this` which identifies the current target object. For example, consider the following client code:

```
BankAccount acc1 = new BankAccount();
acc1.setAccountHolder("Andy");
```

When we call the `setAccountHolder()` method here, the method implicitly receives `acc1` as a hidden parameter. This tells the method it's acting on the `acc1` object (as opposed to some other `BankAccount` object).

Within the `setAccountHolder()` method, we can use `this` explicitly as follows, to make it obvious we're accessing the `accountHolder` member on "this" object:

```
public void setAccountHolder(String ah) {
  this.accountHolder = ah;
}
```

However, it's also possible to omit the `this` keyword as shown below. When the Java compiler sees the unadorned usage of `accountHolder`, it will first see if there's a local variable named `accountHolder`, and if not it will then look at the class definition to see if the class itself defines accountHolder (which it does).

```
public void setAccountHolder(String ah) {
  accountHolder = ah;
}
```

Defining Instance Methods

- A class can define any number of instance methods
  - So-called because each they operate on a particular instance
- General syntax:

```
[public | private | protected] type methodName(params) {
  methodBody
}
```

- Example:

```
public class BankAccount {
  …
  public double deposit(double amount) {
    balance += amount;
    return balance;
  }

  public double withdraw(double amount) {
    balance -= amount;
    return balance;
  }
  …
}
```

BankAccount.java

15

As well as defining getter and setter methods, you will also want to define general business methods in your class. Each method can have a visibility specifier, and must specify a return type and a parameter list.

The example in the slide shows how to define a couple of simple methods to deposit and withdraw money in a bank account. Notice the `return` statement in each method, to return a single result from the method.

Overloading Methods

- A class can contain several methods with the same name
  - As long as the number (or types) or parameters is different
  - This is known as "overloading methods"
  - Useful, because it gives client code several ways to invoke the same semantic behaviour
- Example:

```
public class BankAccount {
  …
  public double deposit(double amount) {
    balance += amount;
    return balance;
  }

  public double deposit(int dollars, int cents) {
    double amount = dollars + cents/100.0;
    return this.deposit(amount);
  }
  …
```

BankAccount.java

- Autoboxing and varargs complicate overloading (see later)

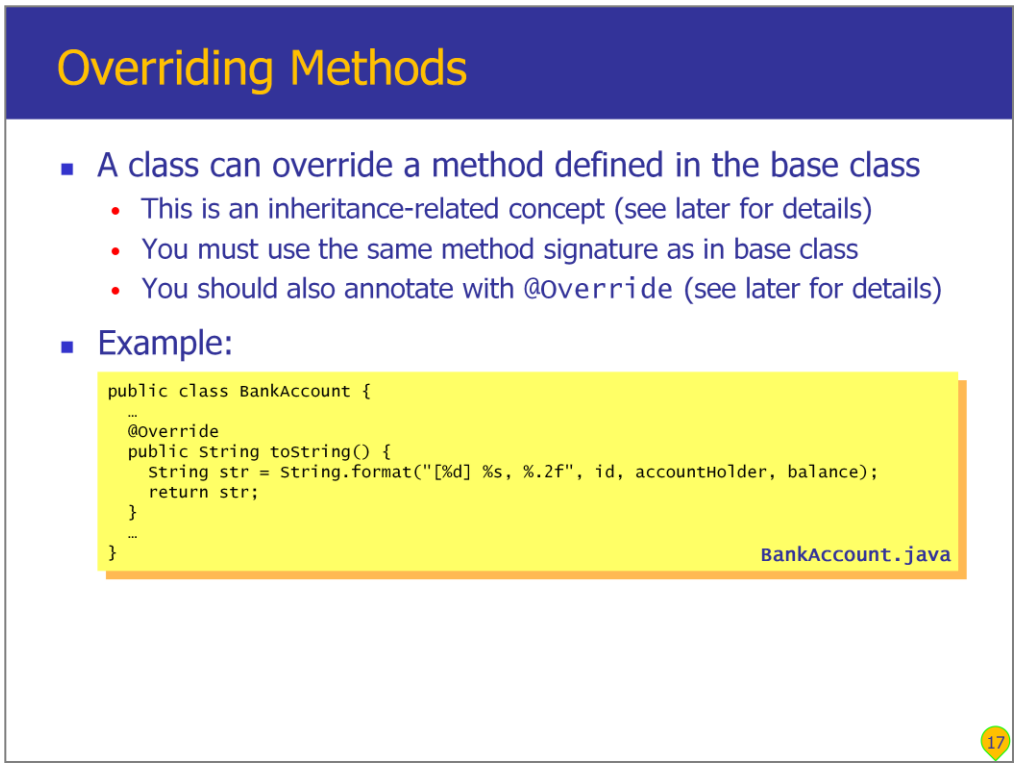
16

Overloading is a useful language feature that allows you to define several versions of a method in the same class, where each version takes a different number of parameters or parameters of different types.

The purpose of overloading is to make it easier for client code to call your methods. The client code can choose which overloaded version of the method to call, depending on what parameters it wants to pass into the method.

Overloading is used extensively in the Java SE library. For example, see the documentation for the `PrintStream` class, which provides many overloaded versions of the `print()` and `println()` methods:

- http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html



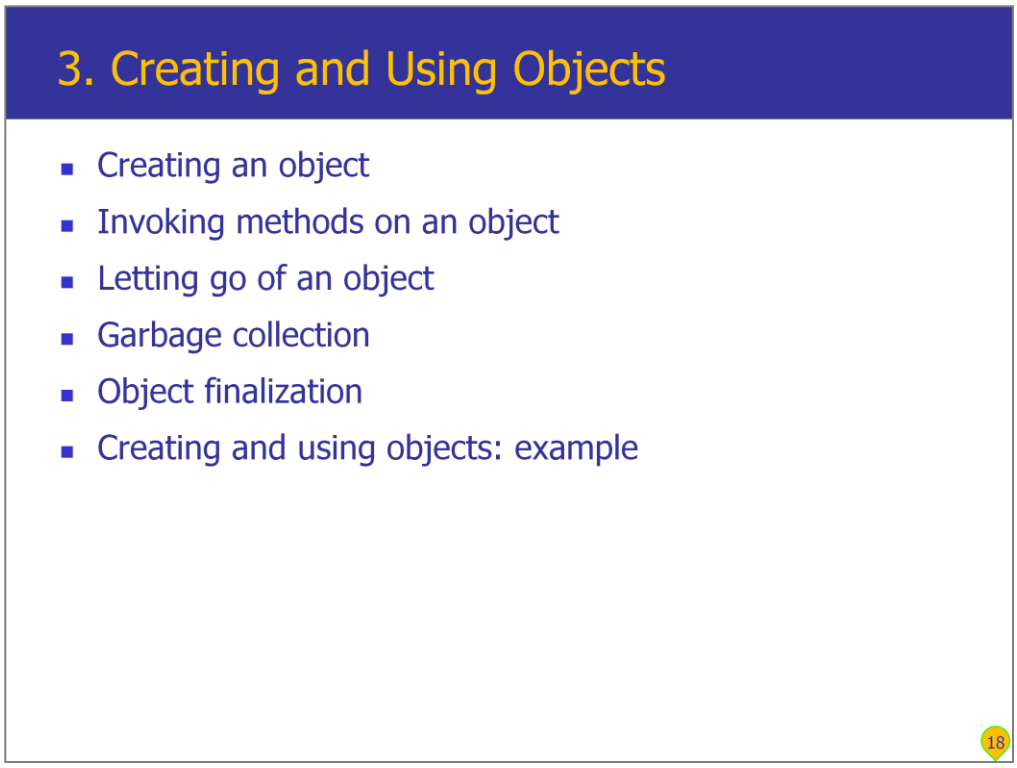
Overriding might sound similar to overloading, but it's an entirely different mechanism. Overriding pertains to inheritance, where a superclass implements a method in a certain way and a subclass wants to offer a different implementation.

We'll discuss method overriding in detail in the Inheritance chapter. However, to whet your appetite, we've shown how the `BankAccount` class can override the `toString()` method. The important fact here is that every single class in Java automatically inherits from a special base class called `Object`, which defines a small number of general-purpose methods such as `toString()`.

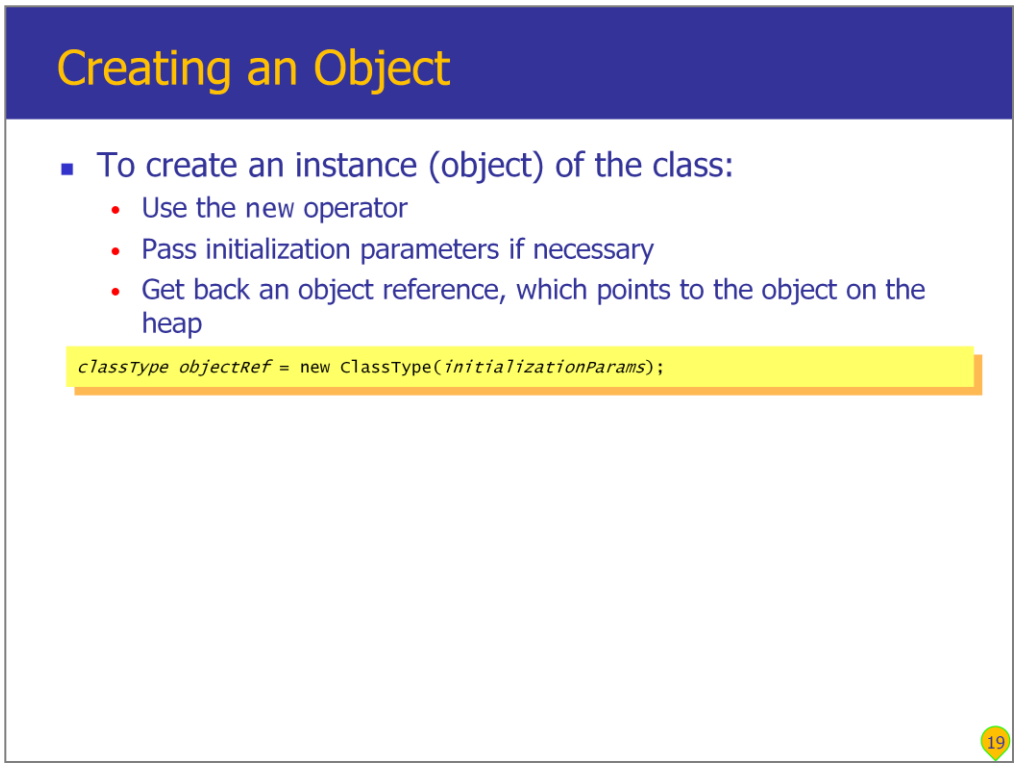
The purpose of `toString()` is to return a textual representation for the current object. The basic version of `toString()`, as defined in the `Object` class, simply returns a string indicating the object's class name, followed by @, followed by the object's hash code. This isn't particularly helpful, so we've chosen to override `toString()` in `BankAccount` to return more useful string that describes the `BankAccount` object's details.

There are a few important rules when you override a method:

- The method must have the same name as the original method in the superclass (obviously!).
- The method must have the same parameter list.
- The method must have compatible return types and exception lists. More details to follow in the Inheritance chapter.
- You should annotate the method with `@Override`, to indicate the fact you're consciously overriding a method from the superclass.

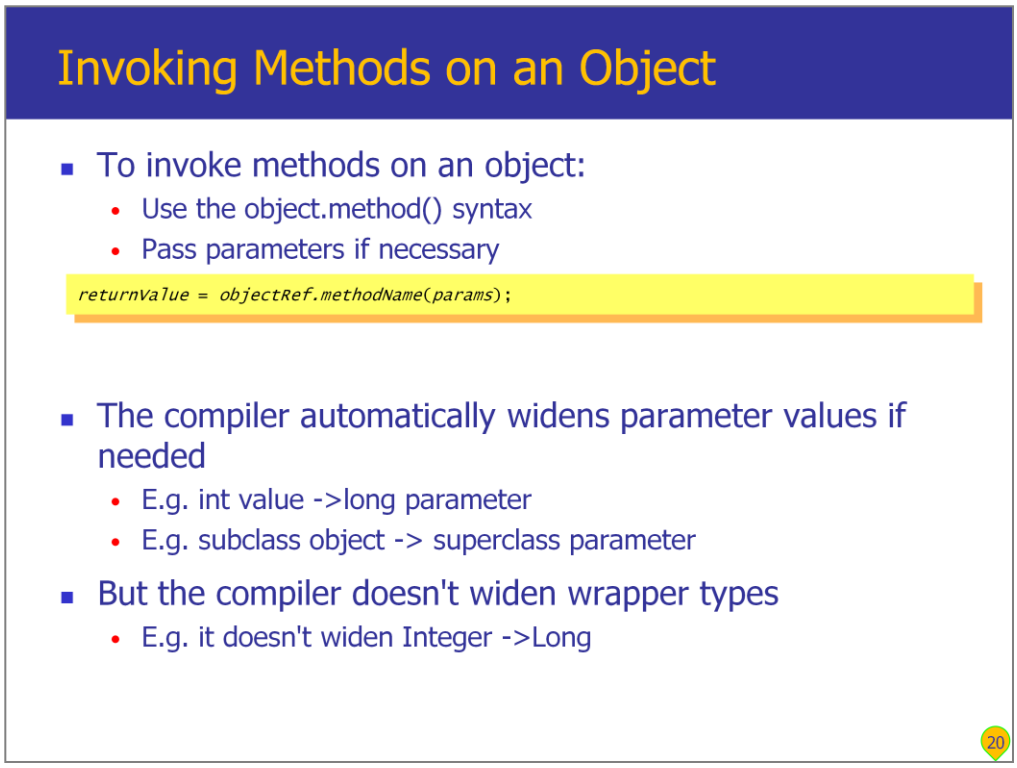


This section goes into detail about how to create and use objects. We'll also consider what happens when an object is no longer needed, and describe the garbage collection mechanism.



To create a new instance of a class - i.e. to create an object - you must use the new keyword. After the new keyword, specify the class type you're interested in, followed by parentheses. The parentheses allow you to pass parameters into the class's constructor, to specify initial values for the object. We'll take a look at constructors and initialization later in this chapter.

The new operator returns a reference to the new object. Typically, you store this reference in a variable of the appropriate type. You use this variable thereafter, to access the object you've just created.

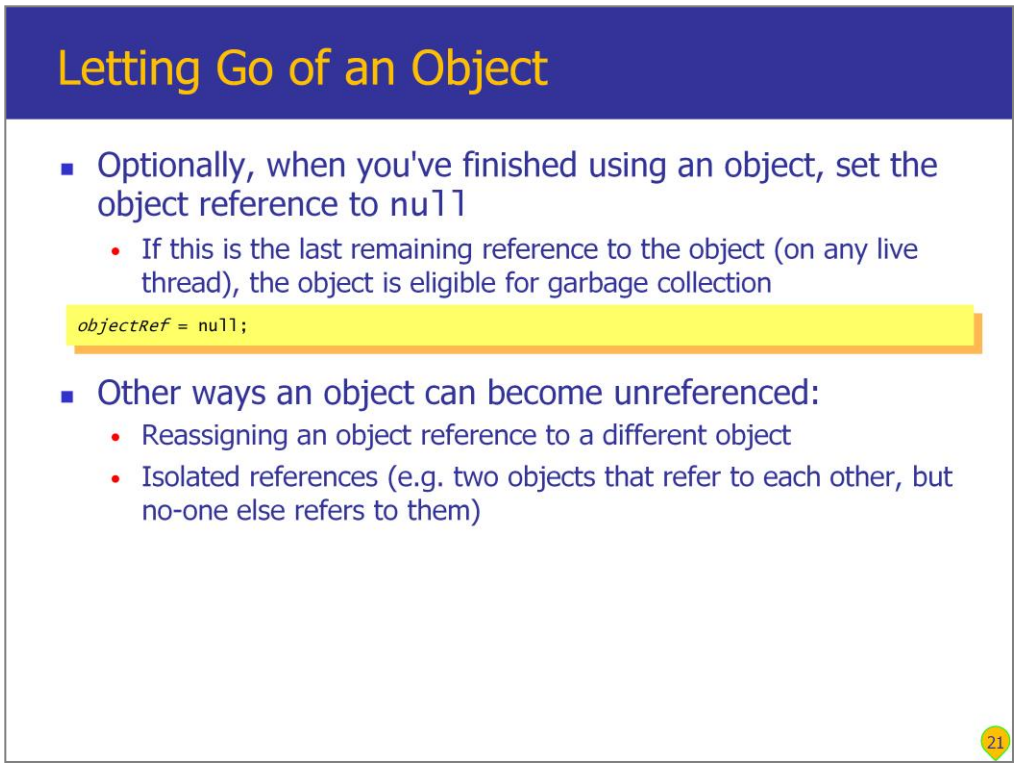


After you've created an object, you can invoke methods on the object via the object reference. The code box in the slide shows the general syntax; you use the dot operator upon an object reference, followed by the name of the method you want to call.

To pass parameters into the method, specify values in the parentheses. If the method doesn't take any parameters, just use empty parentheses.

If the method returns a value, you can capture the variable via an assignment operator, as shown in the code box in the slide. There are other ways to use a return value - for example, if a method returns a boolean, you can use it directly in an `if` statement such as the following:

```
if (acc1.isOverdrawn()) {
  …
}
```



Java doesn't have a "delete" operator (unlike C++). When you've finished using an object, you can just set your object reference to `null`. If this is the last remaining reference to the object in your application, then the object becomes available for garbage collection.

Note that garbage collection is non-deterministic. The garbage collector runs intermittently, so it might be some time before your object is actually garbage collected.

Garbage Collection

- The JVM decides when to run the garbage collector
  - Typically when it senses memory is getting low
  - The GC looks for objects that are eligible for garbage collection, and reclaims their memory
- You can request the JVM to do a garbage collection
  - But Java is so good, you're advised not to do this nowadays

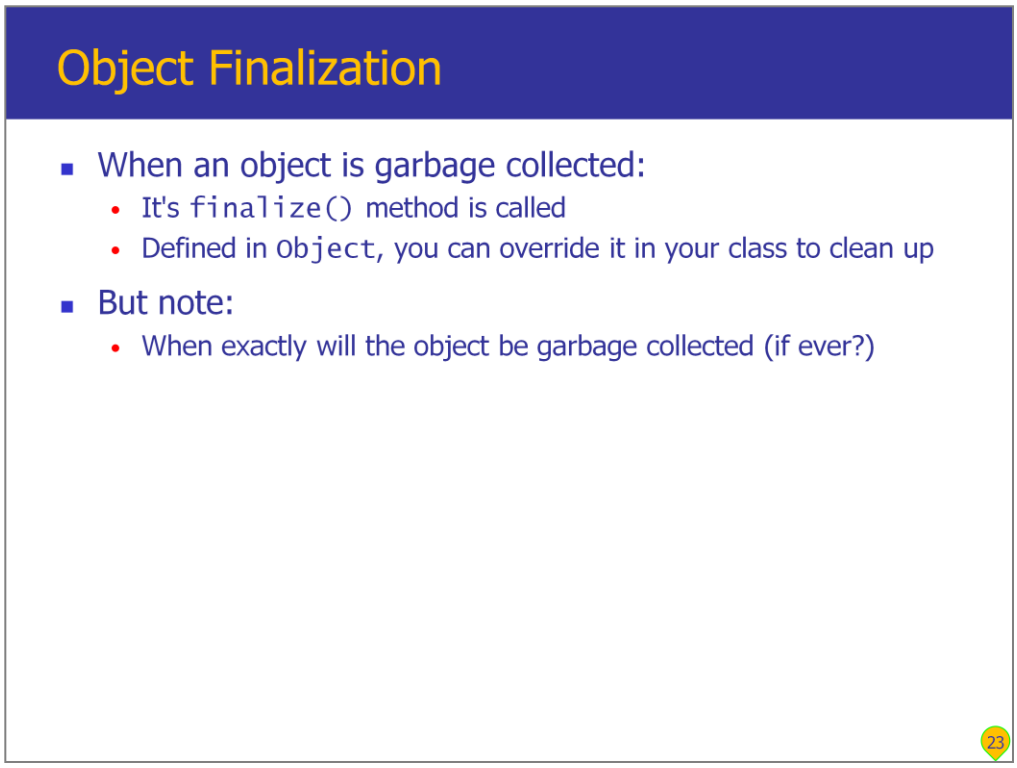
```
System.gc();
```

- If you want to find out how much free memory you've got:

```
Runtime rt = Runtime.getRuntime();
…
rt.gc(); // Alternate to System.gc()
…
System.out.println("After garbage collection, free memory = " + rt.freeMemory());
```

22

This slide shows some advanced techniques for managing garbage collection and for ascertaining the amount of free memory in your application. You'll probably not need to do this very often (if at all) in most Java applications. Specifically, we recommend that you don't call `gc()`.



The `Object` class defines a method named `finalize()`, which is called just before an object is garbage collected. You can override this method in your class, if there is some vital work you need to do before an object disappears.

Before you rush into this, think again about how garbage collection works in Java. You don't know exactly when your objects will be garbage collected; they might lie around in memory for some time before the garbage collector sweeps them up. Only at that point will the finalization code be executed… is this behaviour deterministic enough for your requirements?

Creating and Using Objects: Example

- Discuss the following example:

```
package demo.classes;

public class UseBankAccount {

  public static void main(String[] args) {

    BankAccount acc1 = new BankAccount();
    BankAccount acc2 = new BankAccount("John Smith");

    workWithBankAccount(acc1);
    workWithBankAccount(acc2);

    System.out.println("Next account will have ID " + BankAccount.getNextId());
  }


  public static void workWithBankAccount(BankAccount acc) {

    acc.deposit(100 * Math.random());
    acc.deposit(10, 50);
    System.out.println("\nBalance after deposits: " + acc.getBalance());
    acc.withdraw(30);

    System.out.println(acc.toString());// Or just acc...
    System.out.println("Created: " + acc.getCreationTimestamp());
  }
}
```

UseBankAccount.java

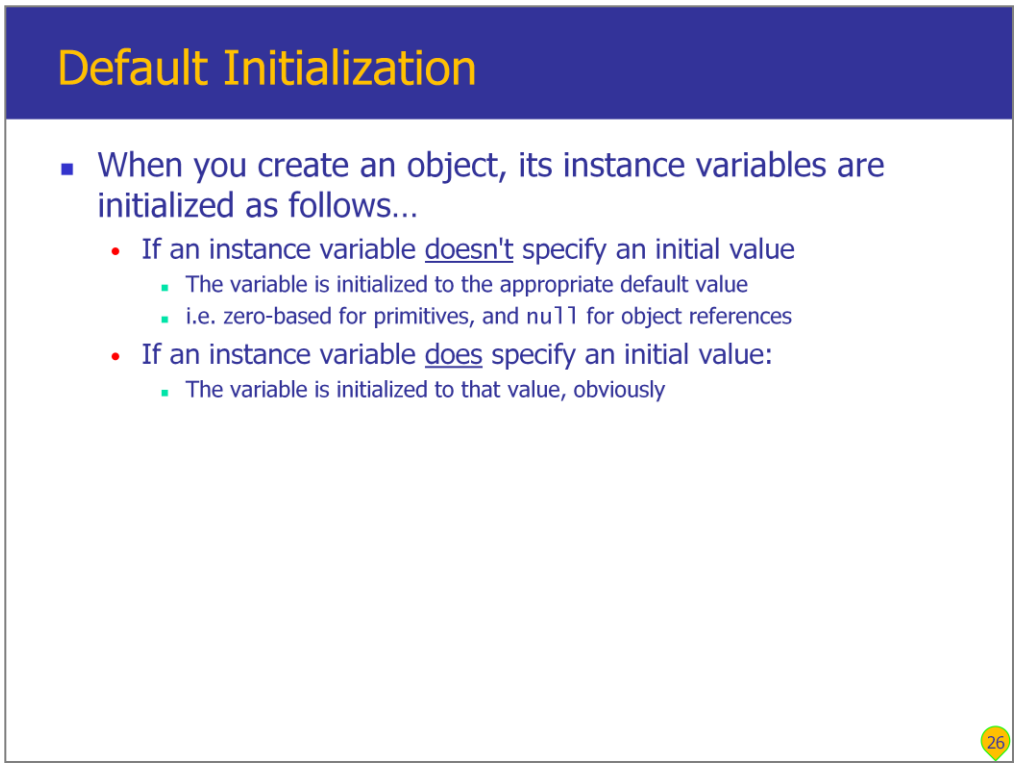
This slide shows a complete example of how to create and use `BankAccount` objects. You can run this example in the demo project, to see what happens.

4. Initialization

- Default initialization
- The role of constructors
- Defining a no-arg constructor
- Defining parameterized constructors
- Constructor chaining
- Initialization blocks

25

This section takes a closer look at initialization. We'll see how to define constructors in your classes, to specify how objects should be initialized. We'll also take a look at some additional useful techniques along the way.



When you create an object, all of its instance variables are guaranteed to be initialized (even if you don't perform any explicit initialization yourself).

By default, numeric types and characters are initialized to zero, booleans are initialized to `false`, and object references are initialized to `null`. For example:

```
class BankAccount {
  private double balance;         // 0.0 by default
  private boolean vipAccount;     // false by default
  private String accountHolder;   // null by default
  …
}
```

If this doesn't take your fancy, you can initialize variables when you declare them. For example:

```
class BankAccount {
  private double balance = 50;    // Opening balance is 50
  private boolean vipAccount;     // false by default
  private String accountHolder;   // null by default
  …
}
```

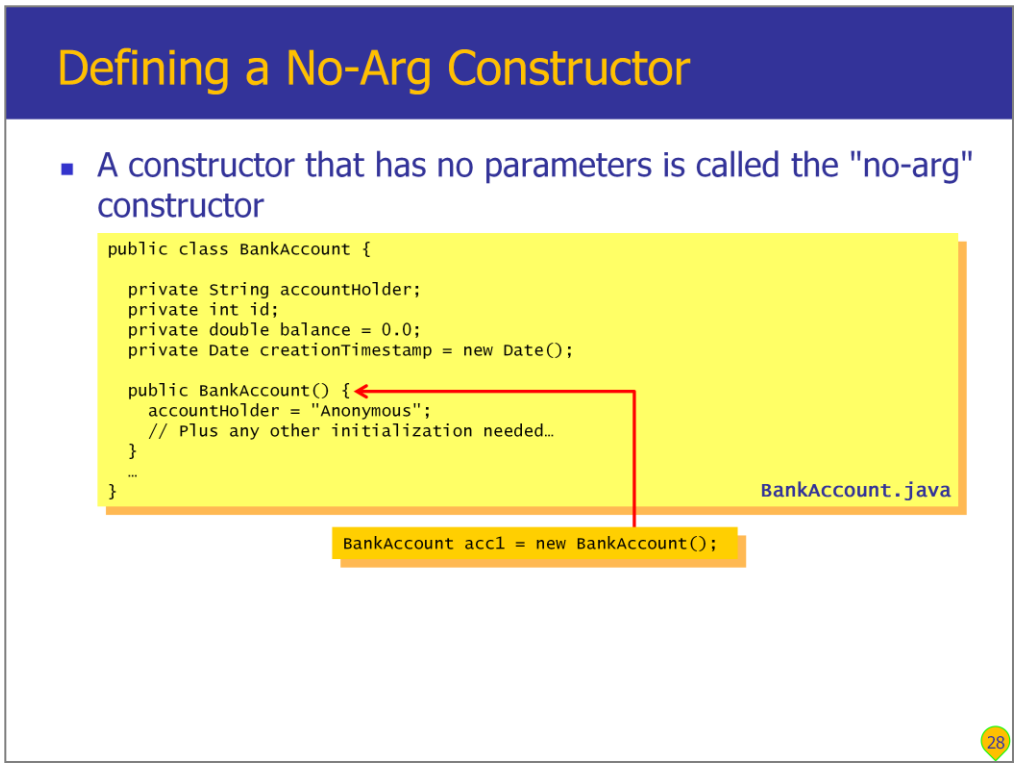
The Role of Constructors

- In addition (or instead) of default initialization…
  - You can define constructors in your class to perform non-trivial initialization
- Rules for defining constructors:
  - A constructor is a method with the same name as the class
  - Typically `public`, to allow client code to access
  - No return type, not even `void`
  - Can specify parameters
  - Cannot be `static`, `abstract`, or `final`

27

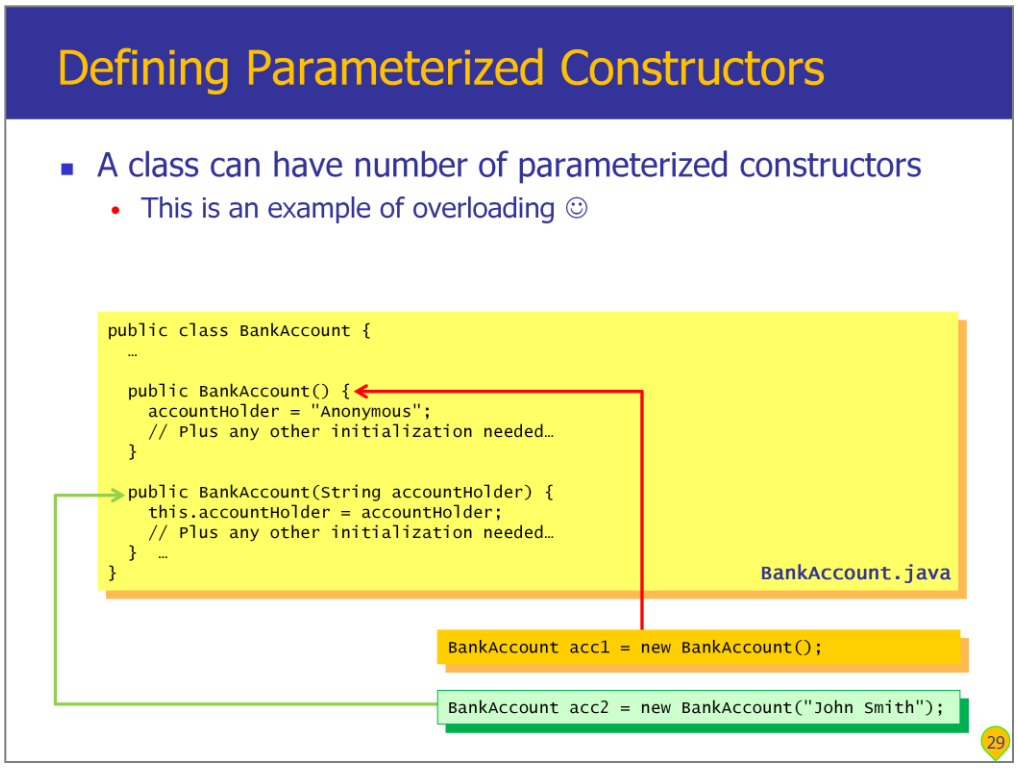
For most classes, you need to provide one or more constructors. A constructor is a special member function that is called automatically when an object is created. The purpose of the constructor is to initialize the object with sensible values for all its instance variables.

There are very strict rules about how to define constructors, as listed on the slide above. Note that you can define any number of overloaded constructors in a class, taking different numbers or combinations of parameters.



The example in the slide shows a simple no-argument constructor for the `BankAccount` class. It also shows how to use this constructor in client code - the constructor is called automatically when you create a new object.

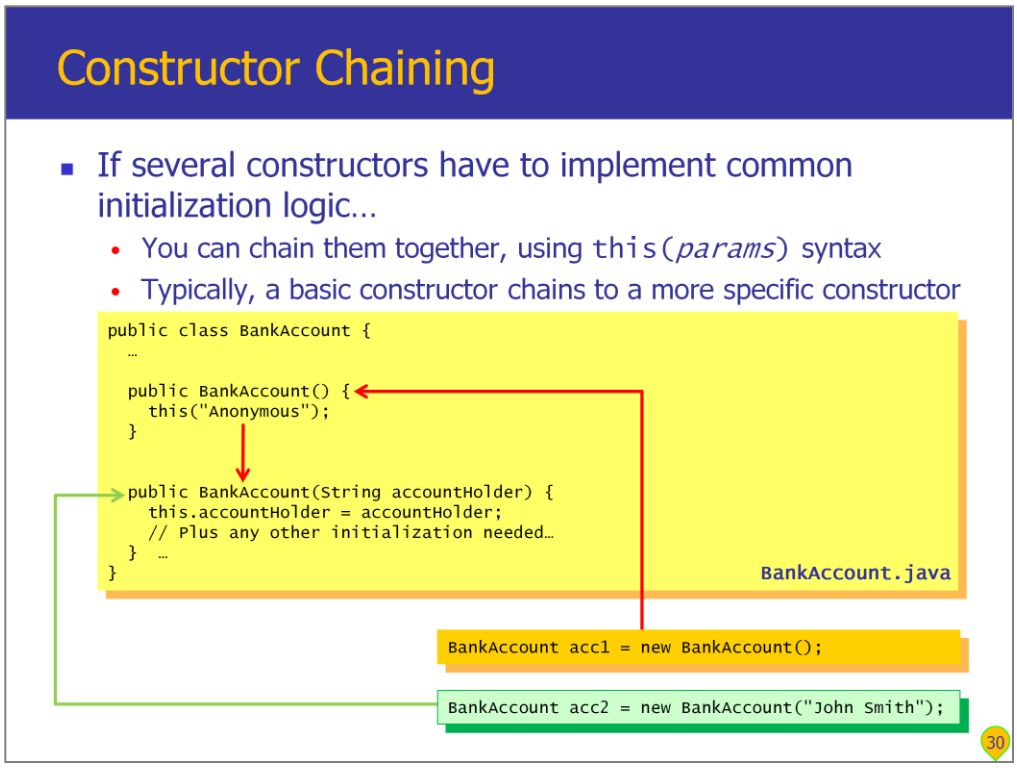
This constructor is obviously a bit too simplistic… it just sets the `accountHolder` instance variable to the fixed string `"Anonymous"`. A more useful constructor would allow you to specify the real name of the account holder as a parameter. The following slide shows how to do this.



You can define any number of constructors in a class. Each constructor must obviously have the same name as the class, but can take different combinations or numbers of parameters.

Consider the example in the slide:

- The first constructor is a no-argument constructor, as per the previous slide. This constructor will be called when you create a new `BankAccount` object without passing any parameters to the constructor.
- The second constructor is a parameterized constructor. This constructor takes a `String` parameter, to specify the real name of the account holder. This constructor will be called when you create a new `BankAccount` object and pass a `String` parameter to the constructor.

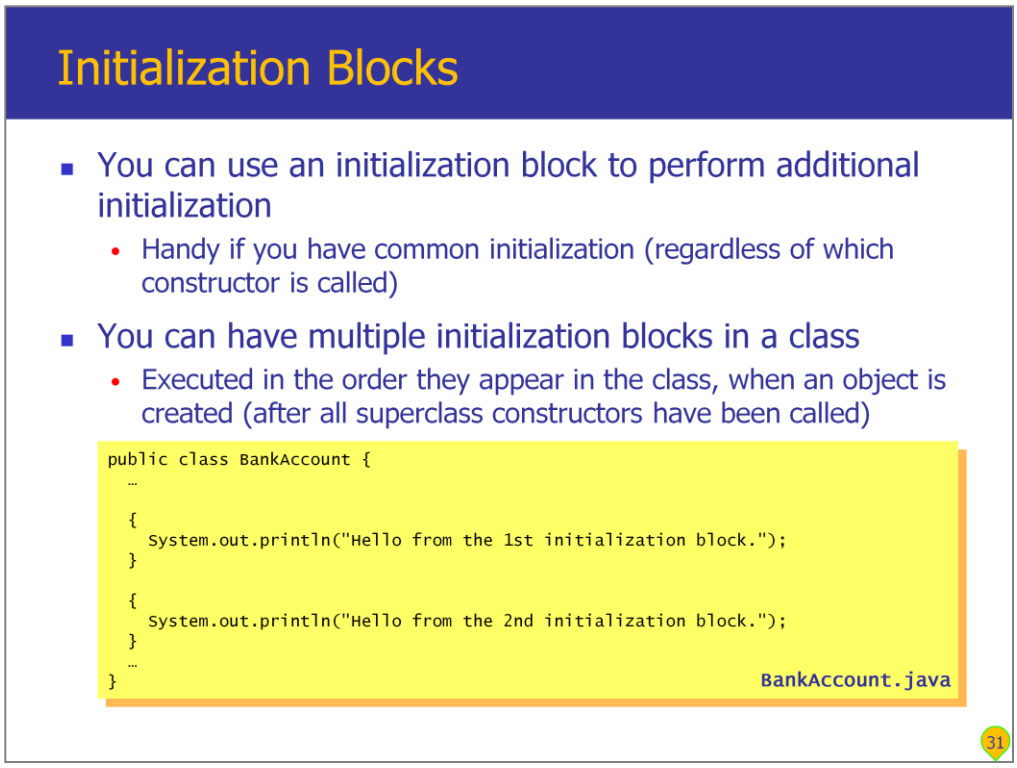


If you define several constructors in a class, you might find that some of the constructors perform common initialization steps. In this case, rather than duplicating the initialization code in each constructor, you can just define the logic in one constructor, and then call that constructor from the other constructors.

For one constructor to call another constructor, use the following syntax:

```
this(parameters);
```

This must be the first statement in your constructor. This technique is known as constructor chaining, and its purpose is to avoid code duplication in your constructors.

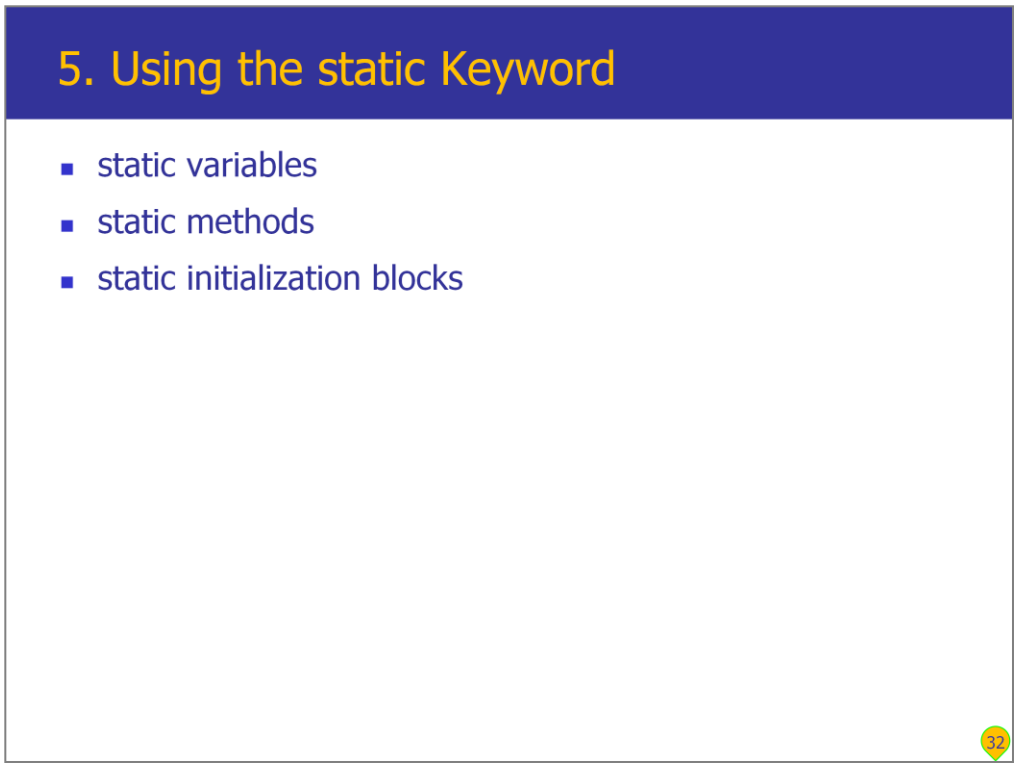


To conclude our discussion on initialization, there is one additional technique you can use to perform initialization - i.e. initialization blocks.

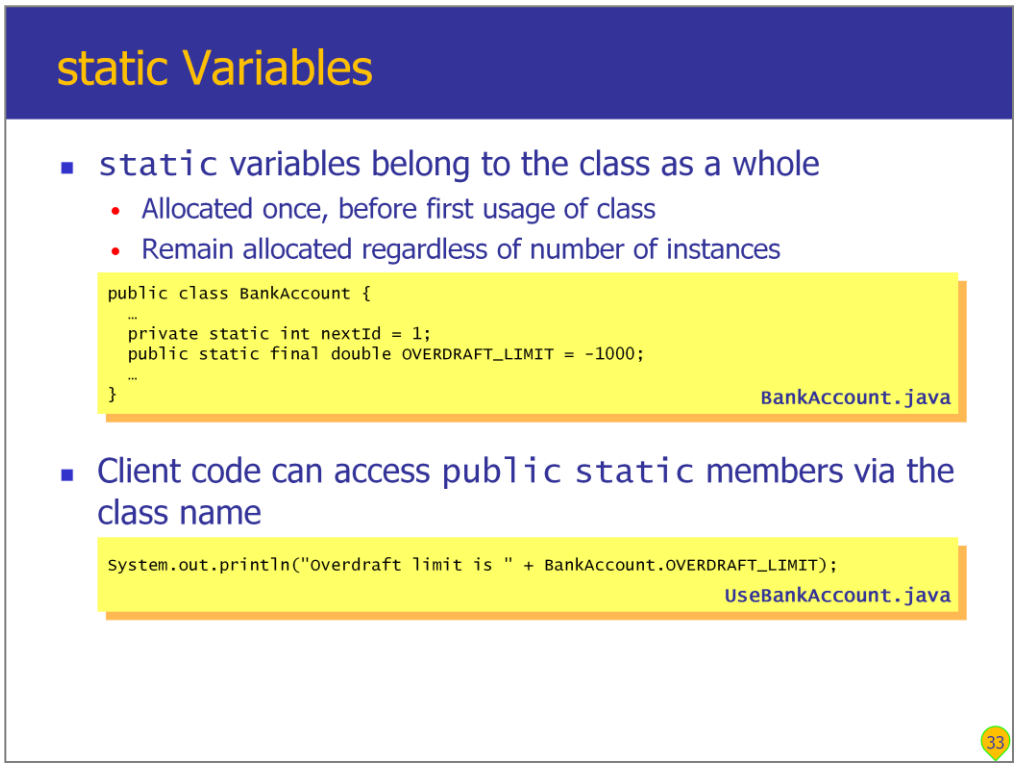
An initialization block is a set of {} with some code inside. It looks a bit like a method, but without a name (or parameters or return type).

You can define any number of initialization blocks in a class. They are executed when you create an object, in the order they are defined within the class. After any initialization blocks have been executed, Java then calls a constructor to complete the initialization of the object.

Note that the use of initialization blocks is a relatively advanced technique, and is rather uncommon in most classes.



This section lifts the lid on the `static` keyword. You've seen this a few times already, e.g. when you define the `main()` method in an application. Now is the time for us to consider what all this means.



When you define a variable in a class, you can prefix the variable declaration with the `static` keyword. This makes the variable a "class variable", which means there will only be a single copy of this variable which will be shared by all instances of the class. Indeed, the `static` variable exists even if you don't actually have any instances of the class.

Common uses of `static` variables are as follows:

- To define a "global" counter for a class, e.g. number of instances.
- To define class-wide values, such as the interest rate for all bank accounts.
- To define class-wide constants, e.g. `MILES_TO_KM` and `KM_TO_MILES`.

If you define a `static` variable as `public`, then you can access it in client code. To access a `static` variable, use the name of the class (rather than using a particular object). For example, the second code box in the slide shows how to access the `OVERDRAFT_LIMIT` variable defined in the `BankAccount` class.

static Methods

- `static` methods implement class-wide behaviour
  - E.g. getters/setters for `static` variables
  - E.g. factory methods, responsible for creating instances
  - E.g. instance management, keeping track of all instances
- Note:
  - A `static` method can only directly access `static` members of the class (i.e. not instance variables/methods)
  - This is because `static` methods don't receive a `this` reference

```
public class BankAccount {
  …
  public static int getNextId() {
    return nextId;
  }
  …
}
```

BankAccount.java

```
System.out.println("Next account will have ID " + BankAccount.getNextId());
```

UseBankAccount.java

34

When you define a method in a class, you can prefix the method definition with the `static` keyword. This makes the method a "class method" rather than an "instance method".

`static` methods don't receive a `this` parameter, so they have no direct access to any particular instance. As a consequence, `static` methods can only directly access other `static` members, which are shared across the entire class.

Common uses of `static` methods are as follows:

- To define getters and setters for `static` variables.
- To implement a factory mechanism, in order to control how instances of the class are created. For example, define a `private` constructor (to prevent client code from creating objects directly), and then define `static` method(s) that create new objects in a controlled way. Note that `static` method(s) are part of the class, so they're allowed to access `private` constructors defined in the same class.
- To house stateless operations that would otherwise be global methods. The `Math` class is a good example of this - it has lots of `static` methods such as `Math.sin()` and `Math.cos()`.

To call a `static` method, use the name of the class (rather than using a particular object). For example, the second code box in the slide shows how to call the `getNextId()` method defined in the `BankAccount` class.

static Initialization Blocks

- You can use a static initialization block to initialize `static` variables
  - Handy if you have non-trivial one-off initialization to perform
- You can have multiple static initialization blocks in a class
  - Executed in the order they appear in the class, when the class is first loaded

```
public class BankAccount {
  …

  static {
    System.out.println("Hello from the 1st static initialization block.");
  }

  static {
    System.out.println("Hello from the 2nd static initialization block.");
  }
  …
}
```

BankAccount.java

35

The third and final usage of the `static` keyword is to define `static` initialization blocks. This is similar to instance initialization blocks discussed earlier in this chapter, but prefixed with the `static` keyword.

`static` initialization blocks are invoked in the order they are defined in the class, at some point before the class is first used in your application code. The initialization runs once only per class, so this is a good place to perform non-trivial initialization for your `static` class members.

Any Questions?

Annex A: UML Diagrams

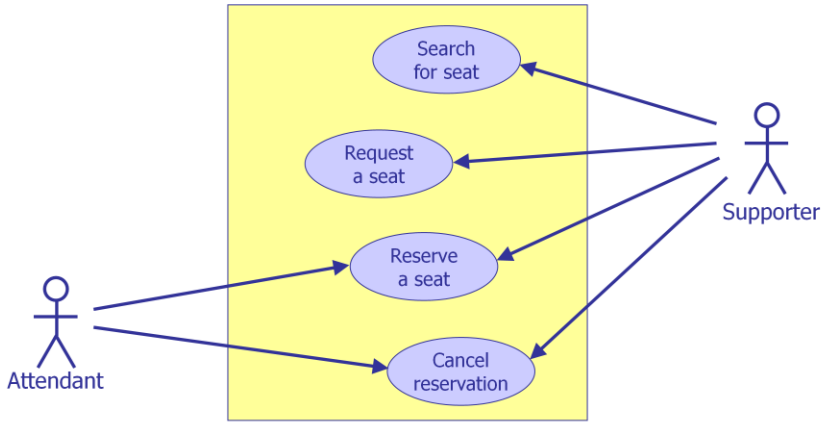
- Use Case diagrams
- Class diagrams
- Statechart diagrams
- Sequence diagrams

37

This section shows some examples of common UML diagrams. These can be a useful way to describe design decisions, even if you don't want to go through a formal OO analysis and design process.

Use Case Diagrams

- Here's a small part of a Use Case diagram for a Ticket Reservation System
  - Allows football supporters to book tickets for a football match

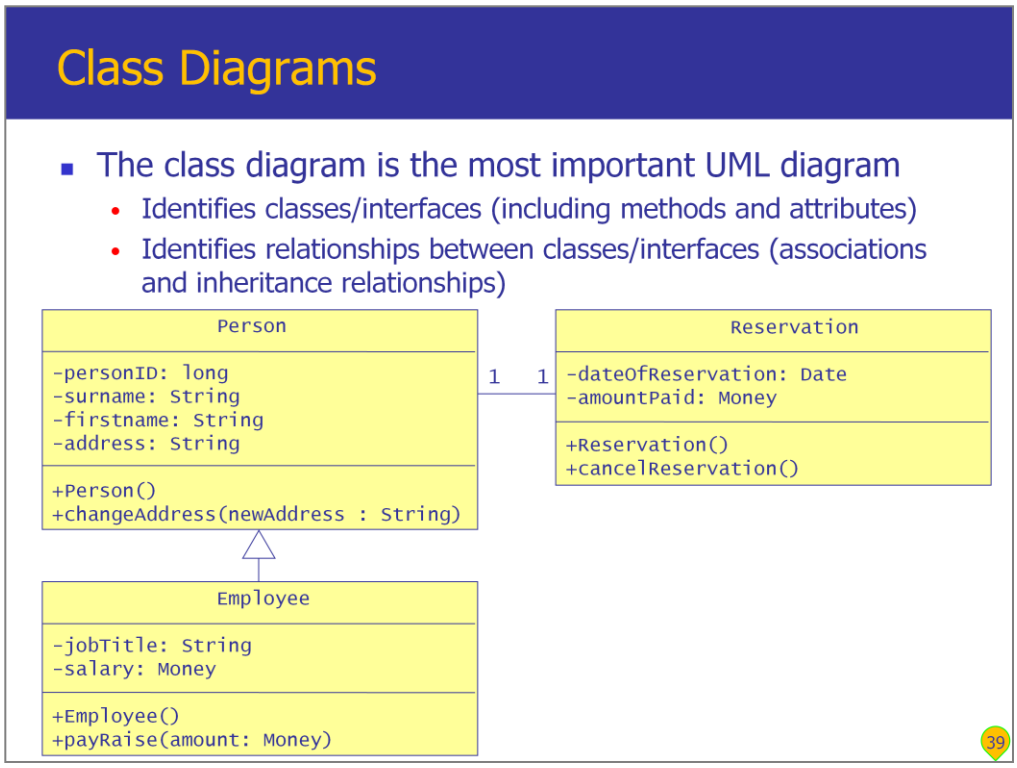


38

This slide shows a Use Case diagram in UML. These diagrams are useful during early OO analysis and design, when you're trying to identify the scope and capabilities of the system you are going to build.

For each use case, define the following information

- Which actor instigates the use case
- The initial state of the system
- The logical interactions between the actor and the system
- Sequential operations, selection, and iteration
- The business rules that must be followed in the use case
- The final state of the system



Each box in a Class diagram is a class or interface...

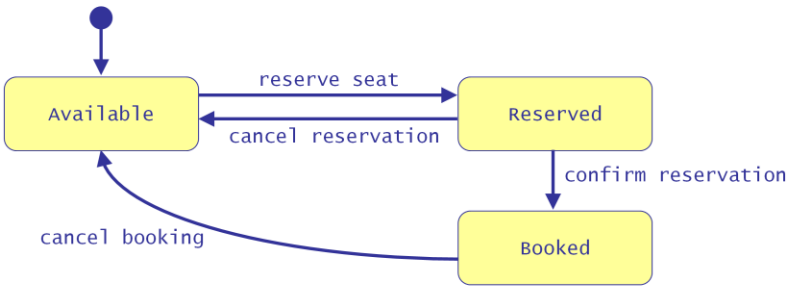
A class defines the data and behaviour for a "kind of thing" in your system:

- The data (a.k.a attributes) is denoted in the middle section of a class box. For example, a person has a person ID (which is a long integer), plus a surname, first name, and address (all strings),
- The behaviour (a.k.a. methods) are denoted in the lower section of a class box. For example, a person has a method named `changeAddress()` plus a strange-looking method named `Person` (this is a special initialization method known as a constructor - more on this later).

An interface is like a class that only specifies method names and signatures. It doesn't define any data. We'll describe interfaces in more detail towards the end of this course.

Statechart Diagrams

- Statechart diagrams show how a particular class behaves depending on its current state
  - Identifies the various states that an object can be in, the events that it recognizes in each state, and how it responds to these events
- For example, here is a statechart diagram for a `Seat` class



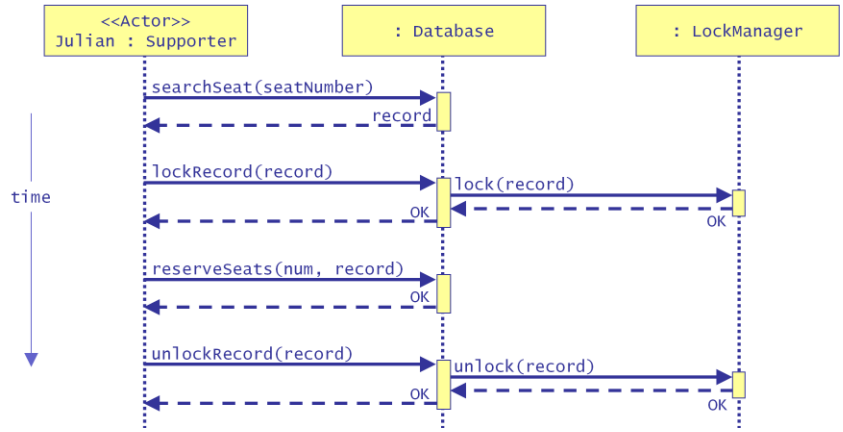
40

Statechart diagrams aren't as important as Class diagrams, but they do have their place. The idea is that you can define a Statechart diagram to describe how a single object's behaviour varies depending on its current state. This is only really needed if an object has dramatically different behaviour based on its current state.

The example in the slide shows a simple Statechart diagram for a `Seat` (e.g. in a football stadium). Each box represents one of the allowable states of a seat. The arrows show the allowable transitions between states.

Sequence Diagrams

- Sequence diagrams show how scenarios unfolds over time
  - Identifies the interactions between objects, and the sequence in which these interactions take place
  - For example, here is a "reserve seat" sequence diagram



41

Sequence diagrams allow you to show interactions between objects over time. They help you to understand the order in which things are going to happen in the system.

Each box along the top represents an object, and the time axis flows from the top of the diagram to the bottom. The horizontal lines indicate messages (i.e. interactions) between objects. The dotted horizontal lines indicate message responses.

Annex B: JavaBean Events

- Overview of events
- Key concepts
- JavaBean listener naming rules

42

This section summaries the "JavaBean" naming convention for defining events in your application. Events are a useful way of telling the client code that something important has happened to your object.

If you've ever implement GUI applications, you'll be familiar with the concept of events. Events are also useful in business applications, to indicate an important business event just occurred - e.g. a stock ticker price exceeded its normal range, a sensor detected an unusual value, etc.

Key Concepts

- Event class
  - Conveys information about an event
  - E.g. information about "action events" (e.g. button clicks) are represented by `ActionEvent`
- Listener interface
  - Specifies methods that will be called on "listener objects", to tell them something interesting happened
  - Listener objects must implement all the methods specified in the listener interface
  - E.g. to listen to "action events", implement `ActionListener`
- Event source
  - Generates events when appropriate
  - E.g. in Swing, the `JButton` class generates "action events" to notify listeners when the button is clicked

44

This slide summarizes some important key concepts for events. Interfaces play a big role with events, so you might want to revisit this topic after we discuss interfaces later in this course.

JavaBean Listener Naming Rules

- The event source (e.g. `JButton`) must provide methods to allow listener objects to register/unregister for events
- Methods to "register" a listener:
  - Name must use the prefix `add`, followed by the listener type
  - Must take a parameter of the listener type
  - Example:

```
public void addActionListener(ActionListener listener);
```

- Methods to unregister a listener:
  - Name must use the prefix `remove`, followed by the listener type
  - Must take a parameter of the listener type
  - Example:

```
public void removeActionListener(ActionListener listener);
```

45

This slide summarizes the recommended approach for allowing client code to register or deregister itself as an event listener. The idea is that once the client code has registered as a listener, the source object will call the client code back whenever the event occurs. We'll see this pattern a lot when we investigate the Swing GUI API later.