Chapter 1

Using Objects

Thinking like a Class Consumer

A View from the Outside

Chapter 1 Using Objects

Goal

To cover the basic concepts and Java code syntax for using objects.

Prerequisites

Basic programming skills.

Objectives

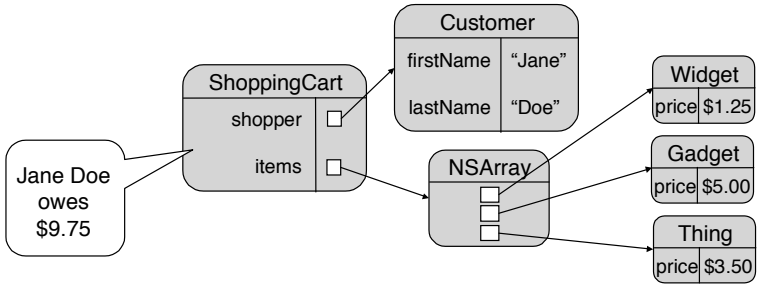
At the end of this lesson, you will be able to:

- Create objects
- Send messages to objects
- Access object attributes
- Collect objects in arrays and dictionaries
- Use casts and determine the class type of an object
- Explain Java naming conventions
- List common coding pitfalls

Objects are the building blocks of Java applications

Objects

- Are "things" that provide behavior and attributes
- Work with other objects–they are often connected together
- Have a life span–they are created and destroyed at run time
- Have a type–each object is an instance of a class
- Play a useful role in your application logic



Objects are the building blocks of Java applications

Java is an object-oriented language. This means that Java programmers think of problems, designs, and code in terms of objects. An object is something that plays a meaningful role in your application. It can be as simple as a number or as complex as a shopping mall.

An object provides useful behavior—it can do things. A shopping cart object can collect items, report which items it currently has, and compute the outstanding balance.

An object has attributes—it has properties or characteristics. A customer object has a name, a credit card number, maybe some coupons.

An object has a life span. You have to create a new shopping cart object when you need it. When you are finished using it, you can throw it away.

Although it is obvious, it is important to see that each object is a specific type of object. An object is clearly a shopping cart, or a customer, or a number, or whatever. By naming its type, you are classifying the object—relating it to other objects that share the same characteristics and behavior. Every object belongs to a specific class of objects.

Objects are connected together to create systems of objects. A customer has a shopping cart that contains several items. In many ways, object-oriented programming is about creating relevant objects and connecting them into meaningful patterns that model a real-world situation.

To get an object, create a new instance of a class

Define a variable—a handle—for keeping a reference to the object

```
ShoppingCart cart;
```

Create a new object and assign it to the variable

```
cart = new ShoppingCart();
```

You can combine both parts

```
Customer shopper = new Customer();
```

With some classes, you can optionally provide initial values

```
Customer shopper = new Customer("Jane", "Doe");
ShoppingCart cart = new ShoppingCart(shopper);
```

To get an object, create a new instance of a class

Before you make an object do something useful, you have to get a hold of it. Unless it already exists, you have to create the object. As long as you need to use it, you need to hang on to it, keeping track of it for future use.

In Java, you first define a variable to reference the object. The variable creates a symbolic name by which you can refer to an object. It is usually called a *reference*. Sometimes this is called a handle or a pointer.

Next, you create a new object and assign it to the reference variable. An object is a real instance of a specific class of objects. Since an object is often called an *instance*, the act of creating a new object is often called *instantiation*.

The Java assignment operator is the equal sign.

You can combine the reference variable definition, the object creation, and assignment to the variable in one Java statement. All Java statements end with a semicolon.

For some classes, you can provide initial data arguments when constructing a new object.

The reference and the object are two different things

Every variable has a type, the *class* of object it references

```
ShoppingCart cart;
```

The class name is pre-defined—the variable name is up to you

The variable is not the object but a *reference* to a potential object

Until you create and assign a new object, the reference is **null**

To create the object, use the **new** keyword and the class name

```
cart = new ShoppingCart();
```

The class name is used like a function and is called a *constructor*

The reference and the object are two different things

The variable declaration has two parts: the type and the name. The type is the class of object the variable will reference. Class names are pre-defined. The variable name is up to you.

How do you know which classes are available? Java defines many standard classes, and development environments like WebObjects add many more. Learning about available classes is a large part of your learning curve. Consult the documentation. You will learn about several useful Java classes in this guide.

Although you have defined a variable, you don't yet have an object. The variable is a reference to an object, it is not an object itself. Some languages call the reference variable a pointer. It is not an object, but something that points to an object. The variable is the name of the object you use when referring to the object in your code. But you still need to create the object itself and assign it to its name.

Until you create and assign an actual object, the reference points to nothing. There is no object. In Java, you say that the reference is **null**. **null** means the non-object.

To create the object, use the **new** keyword followed by the name of class and a pair of parentheses. The syntax looks much like a function call. In Java, this is called invoking a *constructor* because it is a function that constructs a new object, an instance of the specified class.

You can use comments and space in your code

```
/*
  Create a new shopper
  Assign the shopper a new shopping cart
*/

Customer shopper;          // define the references
ShoppingCart cart;

// Create the objects
shopper = new Customer("Jane","Doe");
cart = new ShoppingCart(shopper);
```

You can use comments and space in your code

Java is a free-form language. You can format your code as you wish. You can include blank lines, indentation, break a statement into multiple lines and so on. You may quickly notice common formatting conventions that you should adopt. Conventions allow you to easily understand someone else's code as well as ensuring that they will understand yours. One of the best ways to learn Java is to read and imitate real Java code.

You can include comments in your code. Unlike the strict syntax required for the Java compiler, comments are free-form and intended for human readers. Java provides two different comment styles:

- Arbitrary multi-line comments: begin with /* and end with */
- Single line comments: begin with //. The remainder of the line is considered a comment.

To interact with an object, send it a message

Sending a *message* means invoking an object's *method*

```
cart.discardItems();
```

Some methods take arguments

```
Widget item = new Widget();
cart.addItem(item);
cart.addItemAtIndex(item,0);
```

Some methods return values

```
Customer shopper = cart.shopper();
```

Some methods do both

```
Widget item = cart.itemAtIndex(0);
```

To interact with an object, send it a message

Once you have a reference to an object, you can interact with the object. Objects serve useful roles and your interest is getting them to act. To make an object do something, send it a *message*.

Sending a message is just as simple as it sounds: tell the object what you want it to do. In Java, you send a message with a statement that combines the object name—the reference variable—and the name of the message. They are connected by a period:

```
object.message()
```

This looks much like a function or procedure invocation in traditional languages. In Java, this is called a *method* invocation. You send a message and the object has a method for servicing it. Sending a message is often simply referred to as invoking a method. The difference between message and method reflects the fundamental difference between you and the object. You have a request—the message—and the object has a response—the method.

Some methods take arguments, some return a value, some do both. Many methods neither take arguments nor return values.

What messages can you send to an object? The answer is specific to each class. Just as important as knowing the classes you can use is knowing what those classes do. This is the largest part of your Java learning curve. To understand an object's capabilities, consult the documentation for its class.

Access object attributes using methods

Most attributes are private—you must use an *accessor* method

```
Customer shopper = cart.shopper();        // get
cart.setShopper(shopper);                 // set
```

There are different naming conventions for get accessor methods

```
Customer shopper = cart.shopper();
Customer shopper = cart.getShopper();
```

In some cases attributes are public—you can access them directly

```
Customer shopper = cart.shopper;
cart.shopper = shopper;
```

Access object attributes using methods

Central to the role an object plays is its *state*—its attributes or properties. A customer has a name, a product has a part number, a shopping cart is related to a customer. In Java, these are often called *fields*. In general object-oriented terminology, they are called *instance variables* because each instance of the class has its own set. Most often, objects provide methods to get at these attributes. To get an attribute from an object, send a message.

Methods that access object attributes are common and basic. They have a special name—*accessor methods*. Usually, they come in pairs: one method to get a value, another to set the value. They are often called the get accessor method and the set accessor method respectively. Sometimes, the get method is called the accessor method while the set method is called the mutator method.

Different classes use different naming conventions for the get accessor method. Some methods start with the word “get”, others simply use the attribute name itself, without the word “get”. To use an object properly, consult the class documentation to determine which naming convention it adopts.

In some cases, attributes are directly accessible without sending a message. Java syntax for accessing public attributes is much like accessing fields of a record or a structure in traditional languages. Notice that you do not use parentheses as you would when invoking a method.

You can nest code to avoid creating variables

Create a new object for a method argument

```
cart.addItem(new Widget());
cart.setShopper(new Customer("John", "Doe"));
```

Invoke a method in the return value of another method

```
String name = cart.shopper().lastName();
```

Nest method invocations

```
cart1.addItem(cart2.itemAtIndex(4));
```

You can nest code to avoid creating variables

Often, you need to create an object merely for the purpose of giving it to another object. You don't need your own reference variable in the meantime. Java syntax allows you to nest the code for creating an object within the list of arguments you are sending to another object's method. Create it, pass it, and forget about it—all in one statement.

In a similar spirit, you often need to send a message to get an object then immediately send a message to that object to get what your really after. For example, assume you need the name of the customer that owns the shopping cart. You could create a temporary variable to hold the intermediate object:

```
Customer customer = cart.shopper();
String name = customer.lastName();
```

But in this case, you are interested in the name, not the customer. Java syntax allows you to connect multiple messages into a single expression where each subsequent message is sent to the return value of the previous:

```
String name = cart.shopper().lastName();
```

To generalize, whereever you need to supply an object reference, you can supply any expression that returns an object reference.

Character strings are objects

Character strings are instances of the class *String*

```
String name = shopper.lastName();
```

You can use literal strings

```
String banner = "All widgets on sale";
shopper.setFirstName("John");
```

You can concatenate strings with the "+" operator

```
String name = "John " + "Doe";
Customer s = cart.shopper();
name = s.firstName() + " " + s.lastName();
```

Strings are immutable—once created, you cannot change their value

Character strings are objects

Java represents simple character strings as objects. They are instances of the class named String. Strings are simple values and are used to represent basic object attributes. Because they are so common, convenient handling of strings is built into the Java language itself.

You can use a literal string value in quotes as an alternative to constructing a string object literally. Notice that the following two lines are equivalent:

```
message = "hello";
message = new String("hello");
```

You can combine strings—concatenate them—using the plus operator. This creates a new string instance that combines the values of operands. The String class provides additional methods for manipulating strings. Consult the Java documentation.

Strings are immutable: once they are created, you cannot change their value. If you wish to modify a string without creating a separate result string, you can use the StringBuffer class.

The fact that many objects represent attributes as strings reveals something fundamental about objects: objects are typically *composed* of other objects. A customer is an object. A customer has a name—a string—which is itself another object.

All objects have string representations

All objects have a string representation

```
String debugString = shopper.toString();
```

Concatenation automatically obtains the string representation

```
String debugString = "Customer = " + shopper;
```

You can print strings to your application's standard output

```
System.out.println(debugString);
System.out.println(shopper);
System.out.println("Customer is: " + shopper);
System.out.println("Last name = " +
  shopper.lastName());
```

All objects have string representations

Regardless of its class, any object can generate a string representation of itself. This is useful for debugging and often for displaying a value in a user interface such as a Web page. Every object in Java responds to the message **toString**().

You can print a string to the standard output unit of your application using the **println**() message. Look at the following code:

```
System.out.println("hello");
```

This statement is saying, "send the **println()** message to the **out** object which is available as a public attribute of the **System** object". What happens when you pass an argument that is not a string? Consider the following:

```
System.out.println(shoppingCart);
```

The **println**() method automatically accesses the string representation of the shopping cart object by sending it the **toString**() message. The object generates a string suitable for printing. This also takes place when you use the plus operator to concatenate strings.

```
message = "Customer = " + customer;
```

The customer object is not a String object but the statement automatically obtains a string representation by sending **toString**() to the customer:

```
message = "Customer = " + customer.toString();
```

Java provides non-object primitive data types

Java is a hybrid language—some data are not objects

For efficiency, Java provides *primitive* data types

Many object attributes use primitive types

Many method arguments and return values use primitive types

Manipulate primitive types with built-in operators, not methods

Don't create using **new**—the variable and the value are the same

Java provides non-object primitive data types

Java is a hybrid language—not everything in Java is an object. For efficiency and convenience, Java provides *primitive types* for simple things like numbers, characters, and boolean values.

Even when working with objects, you will need to handle primitive types. They are used to represent many object attributes—the number of items in a shopping cart for example. They are also used for many method arguments and return values.

Primitive types are much like basic data types in traditional non-object-oriented languages. You handle primitive types differently than objects in two fundamental ways:

- Manipulate primitive types with operators not methods
- Don't instantiate primitive types—there is no difference between a value and a reference to the value, they are the same

Useful subset of primitive data types

Type	Contains	Examples
`byte`	8-bit signed value	Any arbitrary bit pattern
`char`	16-bit unicode character	`'a','0', \u00F1`
`int`	32-bit signed integer	`10, -5`
`double`	64-bit IEEE floating point	`10.5, -5.2`
`boolean`	1-bit true or false value	`true`, `false`

Useful subset of primitive data types

Java defines several primitive types. Here is a useful subset: byte, char, int, double, and boolean.

char represents 16-bit unicode characters, not the traditional 8-bit ASCII characters used in languages like C and C++.

int is always 32 bits regardless of the underlying hardware platform. This fixes a number of portability issues inherent in C and C++ due to different word sizes on different machines.

Additional types not shown here offer different possibilities for number values: **short**, **long** and **float**. They differ in size and magnitude, and reflect the C and C++ origins of Java.

Java does not provide any unsigned types.

boolean values use 1 bit and have only two possible values—**true** and **false**. These are Java keywords. Unlike C and C++, Java does not allow numbers or references to be used directly as boolean values. For example, 0 is always the number 0, not the boolean value **false**.

The table shows that literal values are allowed for all primitive types. Whereever you need to supply a primitive type, you can supply a variable, a literal, or an expression that results in a primitive type value.

Useful arithmetic operators for primitive types

Arithmetic operators

- + addition
- - subtraction
- * multiplication
- / division
- % remainder

Arithmetic operators produce a numerical result

Use ( ) for grouping and precedence

Useful arithmetic operators for primitive types

When working with primitive number types, you use operators not messages. Java provides the standard arithmetic operators: +, -, *, /, and %. Java provides several additional operators not shown here such as ++ for increment or += for a combination of addition and assignment. Java also provides itwise operators.

Arithmetic operators expect primitive number operands and produce primitive number results—likewise for any arithmetic expression of arbitrary complexity. Do not confuse primitive number values with either boolean or object values.

There is one exception: the + operator is also valid for concatenating string objects. The result of concatenation is a string object. This is the only case in Java where an operator is overloaded to support object rather than primitive types. This is built into the language. Java does not support operator overloading for custom classes.

You can use parentheses for grouping and readability. Because of the precedence rules in Java, you many need to use parentheses to enforce the meaning of an expression when the default precedence produces unexpected results.

Useful boolean operators for primitive types

Relational operators		Logical operators	
= =	equal to	&&	AND
!=	not equal to	||	OR
>	greater than	!	NOT
>=	greater than or equal to		
<	less than		
<=	less than or equal to		

Relational and logical operators produce a boolean result

Use ( ) for grouping and precedence

Useful boolean operators for primitive types

You can perform a variety of boolean tests using the relational operators: ==, !=, >, <, >=, and <=. The relational operators are valid only for primitive types with the exception of == and != which you can also use for comparing object references (more on this later). The result of an expression using relational operators is a boolean value—true or false.

You can join multiple boolean expressions using the logical operators: &&, ||, and !. The logical operators work only with boolean operands. The result of any boolean expression is a boolean type value, either true or false.

Code examples using primitive types

Variable definitions

```
int count;
double price = 10.75, discount = 0.15;
double total = price * count * (1 - discount);
boolean orderConfirmed = false;
```

Statements

```
count = count + 1;
orderConfirmed = true;
total = cart.total() * (1 - discount);
shopper.setCreditLimit(500.00);
shopper.setCreditLimit(500.00 * 0.75);
shopper.setCreditLimit(cart.total());
```

Code examples using primitive types

You can define variables of primitive types including an initial value. The initial value can be the result of an expression using literals, other primitive variables, and even messages to objects that return primitive values. You can define multiple variables of the same type in one statement. Use the comma to separate the names. If you attempt to use a variable that has not been initialized or assigned, the Java compiler will generate an error.

Java syntax permits great flexibility in building expressions using a mixture of literals, variables, messages to objects, and nested expressions. The key is to make sure the resulting type of each component in the expression matches the overall type, in this case, a primitive non-object type.

You can make decisions for conditional logic

Simple conditional statement with **if** and a boolean expression

```
if (orderConfirmed)
   cart.checkOut();
```

Either-or logic using **else**

```
if (cart.getItemCount() >= 10)
   discount = .25;
else
   discount = .10;
```

Complex boolean expression

```
if (cart.getItemCount() > 0 && !orderConfirmed)
   askForConfirmation = true;
```

You can make decisions for conditional logic

Boolean expressions enable you to make decisions. Often, your code is conditional—you only want to execute it under certain circumstances. For example: if the customer is ready, then send the shopping cart through check out.

Java provides the **if** and **if-else** statements for these occasions. **if** uses a parenthesized boolean expression to determine whether or not the subsequent code should be executed. If the expression is true, the code is executed. If false, it is not. The **else** keyword is optional and provides the alternative choice. If the expression is true do one thing, else, do the other.

The simplicity of the if-else statement is deceptive. Its flexibility permits multi-way decisions of arbitrary complexity:

```
if (subtotal > 1000)
   discount = .20;
else if (subtotal > 500)
   discount = .10;
else
   discount = 0;
```

Java also provides the **switch** statement and a conditional operator for making decisions. These are not shown here.

Multiple statements require a block

Multiple statements within an **if** or **else** clause require a *block*

A block is a group of statements delimited by braces {}

You can create temporary local variables inside a block

```
double total = cart.total();
if (cart.itemCount() > 0) {
   double discount = 0;    // temporary variable
   if (cart.getItemCount() >= 10)
      discount = .25;
   else
      discount = .10;
   total = total * discount;
}
```

Multiple statements require a block

Frequently, you need to do several things based on a certain decision. To group multiple statements within an **if** or **else** clause, you must enclose them in a *block*. A block is a group of statements enclosed in braces. Blocks are used in several different places in the Java language. A block is sometimes called a *scope*.

Within a block, you can define new variables. These are *local variables*, visible only within the block that defines them. They are *temporary variables* in that they come into existence only when you enter the block and they are destroyed when you leave the block.

You can perform basic object tests

Does a variable refer to an object?

```
if (shoppingCart == null)
   // there is no object
```

Do two variables refer to the same object?

```
if (customer1 != customer2)
   // they refer to different customer objects
```

Are two objects equivalent?

```
if (string1.equals(string2))
   // two strings have the same contents
```

You can perform basic object tests

You often make decisions based on simple object tests. The most fundamental test is whether or not a variable refers to an object. Remember that the reference variable is one thing, the object it refers to is another. Java provides the **null** keyword which means "no object". You can use **null** to determine if a variable actually refers to an object. It is illegal to send a message to a variable whose value is null. This makes sense: the variable does not reference a valid object. Notice that you use an operator to make this test, not a message.

In a similar vein, you may have two variables of the same type and wish to know if they refer to the same object. This is called an *identity test* because you are asking if two objects are identical—the same object under different names. Use the comparison operators == or !== to test the reference values.

A third test asks not whether two objects are the same, but whether they are equivalent. They might be different instances of the same class, but have the same contents. Imagine two physical copies of the same credit card. They represent the same account and should be treated equally with respect to making a charge. This is an *equivalence test* rather than an identity test. It requires that you ask the objects themselves using a message rather than an operator. All objects respond to the **equals**() message though how they test for equivalence is specific to each class.

It is a common mistake to confuse identity tests with equivalence tests.

Classes often provide attributes and special objects

Classes often provide attributes, independent of any specific instance

Use a *class* method rather than an instance method

```
int count = ShoppingCart.activeCartCount();
```

Often, class attributes are public—access them directly

```
String version = Customer.Version;
```

To access some objects, use a class method rather than a constructor

```
Store mainStore = Store.headquarters();
```

Note—*Class* methods and attributes are called *static* in Java

Classes often provide constants and special objects

Imagine that you want to know how many active shopping counts there are. How many customers are currently shopping? It doesn't make sense that any particular shopping cart instance would know this information. You need to ask the manager of all shopping carts, the factory that creates them. You may not even have a reference to any particular shopping cart object. But you do know the name of the shopping cart class.

Many classes provide methods and attributes that you can access directly from the class itself. You don't need an object instance. Java calls these *static*—static methods and static variables. Since they are available directly from the class, general object-oriented terminology calls them *class* methods and variables. Because a class is like a factory—it is used to manufacture instances, these are also called *factory* methods.

You can send a message to a class. Use the class name where you would normally use a reference to an object instance. For class attributes, there is one value for all instances, even if there are no instances at all. In Java, you can also send class methods to an object instance. You can get class attributes from the object as well.

A common use of a class method is to access an object instance by a means other than creating a new one directly with the new keyword. For example, instead of creating a new store, ask the Store class for a specific instance—headquarters.

An object has a lifetime—eventually it is destroyed

Java features automatic *garbage collection*

When there are no more references to an object, it is destroyed

When you clear a reference variable, you release the object

```
item = null;
```

When you release an object, it releases its own references

```
customer = null; // customer releases its name
```

All objects are destroyed when your application terminates

An object has a lifetime—eventually it is destroyed

To use an object, you must first create it. What happens when you are done with it? How do you destroy it? This is an important design question since you need to control the amount of resources your application uses. If you create new objects but never destroy them, you are wasting memory and may encounter performance problems.

Java features automatic *garbage collection*. When an object is no longer used, it is automatically treated as garbage and eventually picked up and thrown out—destroyed. How does Java know when you are no longer using an object?

As long as you maintain a reference to an object, the object will not be considered garbage. You can forget about an object—thereby throwing it in the trash—by clearing any and all references you have to that object. The direct way to do this is by assigning **null** to your reference variable.

Many objects are composed of other objects—a customer object consists of at least a string for the first name and another for the last name. When you give up your reference to the customer, you can assume the same for the objects it references—the strings for the name attributes. Although the strings are referenced by the customer, no one has a reference to the customer itself. You can trust that it will all be considered garbage.

Classes types are hierarchical

Object is the root class

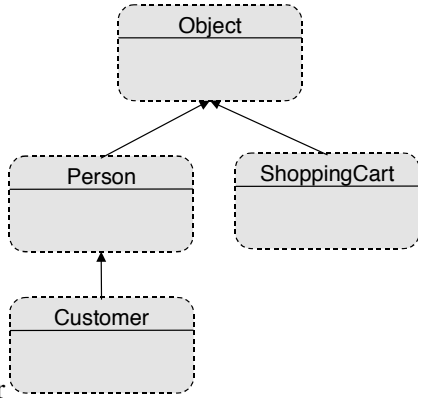
All classes are *subclasses* of Object

ShoppingCart is a kind of Object

Customer is a kind of Object

Customer is also a kind of Person

Person is the *superclass* of Customer



A Customer object is an Object, a Person and specifically, a Customer

Every object is, most generically, an instance of the Object class

Class types are hierarchical

Class types form a hierarchy. It is often called a classification hierarchy because it reflects the way we naturally classify objects in the real world: a class is defined as both similar to yet different from another.

The root class is named Object. It is the most generic class. Other classes are defined relative to the root class—either directly or indirectly, they are subclasses of the Object class. A shopping cart is a kind of object but it is more, it has additional behavior and attributes. ShoppingCart is a subclass of Object; Object is the superclass of ShoppingCart.

The diagram shows a Person class and a subclass called Customer. Generally speaking, a customer is a kind of person. A customer can be treated as a person, and used in any situation where a generic person can be used. But at the same time, a customer is different than a person. A customer is more specifically a customer—it is appropriate in some situations where a generic person is not.

As an object consumer, it is important to see that an object is an instance of a specific class but can be treated more generically as an instance of any of its superclasses. That you can regard an object from multiple perspectives—according to how you classify it—suggests that you can use that object in multiple contexts. How you treat an object—generically or specifically—has impact on how your write your code.

Object references can be generic or specific

This code is correct, though the variable type is not specific

```
Object customer = new Customer();
String string = customer.toString();
```

This code is not correct—the variable type is not specific enough

```
Object customer = new Customer();
String string = customer.firstName(); //error!
```

`firstName()` is a method in the Customer class, not the Object class

Generic references are useful when specific details are unimportant

- A mailing list references all Persons, not just Customers
- A shopping cart references objects of many different classes

Object references can be generic or specific

Sometimes you can treat an object generically. A customer object, like any object, can produce a string representation of itself. Any subclass of Object responds to the **toString**() message. It is not important to know that the customer is specifically an instance of the Customer class. You can refer to it as a generic object. You can declare a reference variable using a generic type—like Object—and use it with any subclass of that type—like Customer. It is valid to assign an object of a specific class to a reference variable of a more general class type.

The class type you declare for a reference variable is a promise you make regarding how you will use the object. You promise to send only those messages defined by the generic class even though the specific object may have additional capabilities defined in its subclass. With a reference of type Object, you can only send messages defined in the Object class. Even though you assign a customer to the reference, you promise to treat it like any other object. You cannot use the reference to ask for a customer's first name. Your code will not compile.

There are many cases where it is useful to treat many different specific types of objects as a single more generic type. A shopping cart can hold many different types of products. But for the purpose of calculating the balance, the shopping cart need only treat each object as a generic product with a price. A company mailing list needs only the name and address of a recipient, without making finer distinctions about which are customers, employees, friends, etc.

Casting allows you to be more specific

To refer to an object using a more specific class, use a *cast*

```
Object anObject = lostAndFound.firstItem();
Customer shopper = (Customer)anObject;
String name = shopper.firstName();
```

You can use a temporary cast to avoid creating a new variable

```
String name = ((Customer)anObject).firstName();
```

A shopping cart holds many classes of objects

```
Widget item = cart.firstItem();     // incorrect
Object item = cart.firstItem();    // correct
// If it is a Widget, use a cast to treat it so
Widget item = (Widget)cart.firstItem();
```

Casting allows you to be more specific

Sometimes, you need to be more specific about an object that is otherwise treated generically. This is often the case when you don't create the object but you get it from another object.

Imagine a lost-and-found object. It collects objects of various types: car keys, wallets, even customers themselves. It is not important to know the specific class of each object, merely that each is simply an object. You can query the lost-and-found using simple messages like **firstItem**() or **nextItem**(). Each message returns a generic object reference valid for any specific class of object.

Once you get the object from the lost and found, you want to make distinctions according to the specific class of object you have. If it is a customer, you can ask for its name. If it is a wallet, you need to check it for identification. To treat an object as a specific type, you must create a reference of that specific type. To assign a generic object to a more specific reference, you explicitly declare your intentions. In Java this is called a *cast*.

A cast must reflect reality: if the object's actual class does not match the class named in the cast, it is a runtime error. The object's class must be the same as or any subclass of the class named in the cast.

You can use a cast to send a message without assigning the object in a reference variable. You can temporarily treat the object more specifically . Because of precedence rules in Java, a nested cast requires an extra set of parentheses: cast first, then send the message.

You can determine an object's class

Testing if an object belongs to a class or any of its subclasses

```
if (anObject instanceof Person)
   // anObject is a kind of Person
   // Could be, more specifically, a Customer
```

Using an object with a more specific class type

```
if (anObject instanceof Customer){
   Customer customer = (Customer)anObject;
   String customerCode = customer.code();
}
```

Checking for the exact class, excluding potential subclasses

```
if (anObject.getClass() == Customer.class)
```

You can determine an object's class

Since an incorrect cast generates a runtime error, you may need to check the actual type of an object before applying the cast. Sometimes, you don't know what kind of object you have—you need to ask it. In Java, this is called *runtime type identification* or RTI.

Java defines the **instanceof** keyword to check if an object is an instance of a specified class. Notice that it is an operator, not a message. Also notice that the 'o' is not capitalized. **instanceof** is true if the object is an instance of the specified class or any of its subclasses.

If you need to check for the exact class type of an object, you can ask the object for its class then compare it—using the comparison operator—against a specific class. Notice the Java construct for getting the class from a class name:

```
Customer.class
```

It is not a message; there are no parentheses. You cannot simply use the class name alone. Using the class name, you can obtain the class itself. This reveals that, in Java, classes are also objects that you can access at runtime.

You can gather objects in a collection

You need a way to group objects together in a *collection*

Java provides many different collection classes, for example

- Vector
- Hashtable

With WebObjects, you use analogous Apple foundation classes

- NSArray
- NSDictionary

These are fundamental for connecting objects into larger structures

You can gather objects in a collection

Until now, you have created explicit references for every object. This is only useful if you know exactly how many objects you need. Often, you don't know the number of objects you will create until runtime. Instead of a single reference to a specific object, you need a single reference to a set of objects. For example, a shopping cart can contain an arbitrary number of items from zero, to one, to many. At runtime, you need to dynamically add and remove objects yet refer to the entire collection with one reference—the shopping cart reference.

For this purpose, Java provides utility classes called *collection* classes. There are different classes for different purposes. The Vector class works much like a traditional array: it collects objects and assigns them numbered positions like 0, 1, or 25. A Hashtable also collects objects but stores them using symbolic lookup keys like strings such as "name", "rank", and "serial number".

WebObjects also provides collection classes. They are part of the WebObjects foundation framework. NSArray is much like a Vector, NSDictionary is much like Hashtable. In WebObjects programs, you most often use the WebObjects collection classes.

Collection classes are essential for connecting objects into larger structures. They allow you to group an arbitrary number of objects determined dynamically at runtime. With collections, you treat many objects as one: you reference the collection object rather than the objects within the collection.

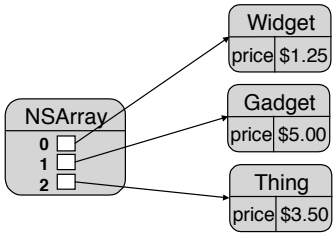
NSArray maintains an ordered collection

NSArray maintains an ordered collection of objects

You access objects by their integer index—starting with 0

NSArray provides runtime bounds checking

NSArray is similar to Java's Vector



NSArray maintains an ordered collection

NSArray objects maintain an ordered collection of objects. Much like a traditional array in many different languages, NSArray stores objects using numbered positions. You access an object in the array using an index of type int. Index values start at 0. If the array holds 10 objects, they are stored with index values ranging from 0 to 9.

Array objects do not have a fixed size. They dynamically grow, automatically, to accommodate new objects. They also provide bounds checking at runtime. If you try to access an object using an index that is out of bounds—less than 0 or greater than the highest valid index, the array generates a runtime error by throwing an exception.

An NSArray object does not really contain the objects themselves, it contains references to the objects. The objects are properly "outside" but accessed by references inside the array.

NSArray is similar to Java's Vector. Note that Java also defines native arrays which are different from either an NSArray or a Vector. There are at least two important differences between native arrays and collection classes:

- Native arrays gather values of the same type; collections can gather a heterogeneous set of object types.
- Native arrays can reference primitive types; collections reference only objects

NSArray and NSMutableArray

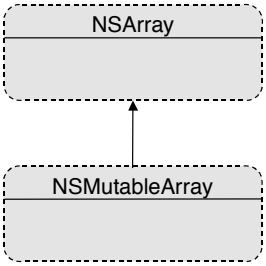
NSArray is constant—you cannot add or remove objects

NSMutableArray can be modified—you can add and remove objects

NSMutableArray has a dynamic size—it grows automatically

NSMutableArray is a subclass of NSArray

An NSMutableArray is a kind of NSArray



NSArray and NSMutableArray

The WebObjects foundation defines two different array classes: one that is constant, one that is mutable. An NSArray is constant: once it has been created with an initial set of objects, you cannot add or remove objects. This is useful and efficient for sharing the array with other clients with the confidence that the array cannot be modified.

NSMutableArray is a subclass of NSArray. It is a kind of NSArray and can be treated generically like any NSArray. But is more specifically a mutable array: you can remove elements and add new ones. NSMutableArray is not constant. NSMutableArray responds to every message that NSArray does. It also responds to additional messages for adding and removing objects. When researching NSMutableArray, be sure to consult the NSArray documentation as well.

NSArray—useful methods

You often get an array from another object

```
NSArray items = shoppingCart.allItems();
```

Getting the current count of objects in the array

```
int count = items.count();
```

Getting an object at a specific index

```
Object anObject = items.objectAtIndex(i);
Widget widget = (Widget)items.objectAtIndex(i);
```

Searching for an object

```
if (items.containsObject(widget))
  i = items.indexOfObject(widget);
```

NSArray—useful methods

You often get a pre-constructed NSArray from other objects. For example, a shopping cart might define a method to return all items in an NSArray. NSArrays are constant: you cannot add or remove objects, but you can access the existing objects.

You can find out how many objects are in the NSArray. You retrieve an object using an index value—and integer. You can ask an NSArray if it contains a specific object and if so, retrieve its index value. If you attempt to retrieve an object using an invalid index, the NSArray will generate an out of bounds exception.

NSArray access methods are defined to return a generic object reference. If you need to treat an object from an array more specifically, you must use a cast. Unlike many arrays in traditional languages, NSArray can store objects of any type, they do not all have to be the same class of object. NSArray can only store objects, not primitive types like int or double. You cannot store a null reference.

The NSArray class defines many additional methods. The methods shown here comprise a useful subset. Consult the WebObjects foundation documentation for details.

NSMutableArray—useful methods

Constructing a new mutable array

```
NSMutableArray items = new NSMutableArray();
```

Adding an object

```
items.addObject(widget);
```

Removing an object

```
items.removeObject(widget);
```

Creating a new mutable array from an existing immutable array

```
NSArray items = shoppingCart.allItems();
NSMutableArray items2 =
   new NSMutableArray(items);
```

NSMutableArray—useful methods

NSMutableArrays respond to the same messages as NSArrays. You can determine the count of objects in the array, get an object at a specific index, and search for an object to determine its index.

NSMutableArray also defines methods for adding and removing objects. When you add an object, it is placed at the end of the array at the next available index. When you remove an object, the array adjusts the indices of all objects that follow, essentially shifting them down to fill in the gap. There are additional messages for inserting and removing an object at a specific index.

You construct a new mutable array like you construct any Java object. The new array is initially empty. Its count is 0. There are no valid indices since there are no objects in the array.

What if you need to add or remove the objects in an immutable NSArray object? You can construct a new NSMutableArray and initialize it with objects from the existing NSArray. Now you can add and remove objects using the mutable array. Although you now have two different arrays you do not have multiple copies of the objects they reference. Arrays contain references not objects. You merely have multiple references to the shared, underlying objects.

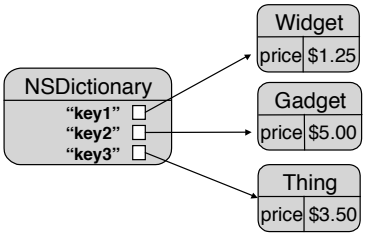
NSDictionary maintains a set of key-value pairs

NSDictionary maintains a collection for efficient lookup

You access an object using a key—another object

The key can be any type of object but it is usually a String

NSDictionary is similar to Java's Hashtable



NSDictionary maintains a set of key-value pairs

Like an NSArray, an NSDictionary maintains a collection of objects. But objects are not stored using numerical indices. There is no implied ordering in an NSDictionary. Rather, objects are associated with keys. You access an object using its key—another object. Usually the key is a String object where the string value is a meaningful symbol like "name", "rank", or "serial number". Dictionaries in other languages are often called associative arrays or hashtables.

NSDictionaries are useful for collecting objects that need to be efficiently accessed using a symbolic lookup key. In this sense, they are like real-world language dictionaries: you supply the word—a lookup key—and the dictionary returns the definition—the object value associated with the key. Dictionaries are implemented for efficient lookup operations. Given a key, they can quickly locate the corresponding value. To do this, dictionaries use a hashing mechanism making them similar to Java's Hashtable class.

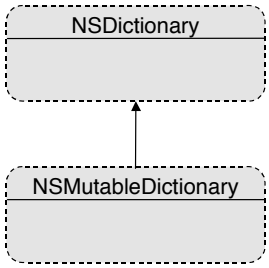
NSDictionary and NSMutableDictionary

NSDictionary is constant—you cannot add or remove objects

NSMutableDictionary can be modified—you can add/remove keys

NSMutableDictionary is a subclass of NSDictionary

NSMutableDictionary is a kind of NSDictionary



NSDictionary and NSMutableDictionary

Like NSArray, NSDictionaries are implemented in two different classes, one constant, the other mutable. NSDictionary is constant: once the dictionary is created, you cannot add or remove objects.

NSMutableDictionary is a subclass of NSDictionary. It is a kind of NSDictionary and can be treated generically like any NSDictionary. More specifically, NSMutableDictionary extends the NSDictionary superclass with additional methods for adding and removing objects. When researching NSMutableDictionary, be sure to consult the NSDictionary documentation as well.

NSMutableDictionaries do not have a fixed size. You can add new objects and the dictionary automatically grows in size to accommodate them. NSDictionaries do not have a concept of bounds checking either. If you ask for an object using a key that is not in the dictionary, the dictionary simply returns **null** to indicate that there is no such object.

NSDictionary—useful methods

You often get a dictionary from another object

```
NSDictionary props = customer.properties();
```

Getting the current count of objects in the dictionary

```
int count = props.count();
```

Getting an object value using a key

```
Object name = props.objectForKey("name");
String serialNumber =
  (String)props.objectForKey("serialNumber");
```

If there is no object for that key, the dictionary returns *null*

```
String rank = (String)props.objectForKey("rank");
if (rank != null)
   // dictionary contains value for "rank"
```

NSDictionary—useful methods

You often get a pre-constructed dictionary from another object. Imagine that a shopping cart can report a set of properties about itself—model name, serial number, the vendor that built it.

You can determine the count of objects in a dictionary. You can get individual objects—values—from the dictionary if you know the correct key. Most often, dictionaries use string objects for keys. Values in the dictionary are key-value pairs: with the right key you can get the associated value.

Like array methods, dictionary methods are declared to return generic object references. Dictionaries can hold any kind of object. A single dictionary often collects object values of several different class types: strings, dates, numbers, and even custom classes like customers and shopping carts. A generic object reference is valid for any class type. When you get an object from a dictionary, you typically use a cast to treat it as a more specific class type.

What if the dictionary does not contain a value for the requested key? The dictionary returns **null**. When in doubt, check the return value. Because of this convention, you cannot store a **null** in a dictionary.

NSMutableDictionary—useful methods

Defining and constructing a new mutable dictionary

```
NSMutableDictionary items =
        new NSMutableDictionary();
```

Adding an object for a key

```
items.addObjectForKey(widget,"product");
```

Removing a value

```
items.removeObjectForKey("product");
```

Creating a new mutable dictionary from an existing dictionary

```
NSDictionary props = shoppingCart.properties();
NSMutableDictionary props2 =
   new NSMutableDictionary(props1);
```

NSMutableDictionary—useful methods

You can construct a new NSMutableDictionary. Initially, it contains no key-value pairs. Its count is 0. Any attempts to retrieve a value will return **null**.

You can add a new object to the NSMutableDictionary, associating it with the specified key. If an object is already stored in the dictionary for that key, it will be replaced by the new object. You can explicitly remove an object associated with a specific key. Subsequent attempts to get an object for that key return **null**.

In some cases, you will want to create a mutable version of an immutable dictionary. You can construct a new mutable dictionary, providing the existing dictionary as an argument to the constructor.

Like NSArrays, NSDictionaries do not really contain objects, only references to objects.

To use a Java class, import its package

Classes are grouped into libraries or *packages* of related functionality

There are many classes in many different packages

- Packages that are part of the Java runtime
- Custom packages from 3rd parties or your organization

To use any class in your code, you must *import* its package

NSArray and NSDictionary are in the Apple Foundation package

```
import com.apple.yellow.foundation.*;
```

The java.lang package is automatically imported

java.lang includes basic classes like Object and String

To use a Java class, import its package

The Java runtime environment defines a large number of standard classes that you can use to build your applications. Products like WebObjects define even more. Your own organization may define its own set of reusable classes.

In Java, classes are organized into *packages*. A package groups related classes that address a specific set of problems. One package might provide advanced math operations and extended value classes. Another package provides classes for performing file I/O. Yet another package deals with networking. WebObjects adds a package for building web-based applications, a package of useful foundation classes like arrays and dictionaries, and a third package for database connectivity—Enterprise Objects Framework.

In general, to use a class in your code, you must explicitly *import* the package that defines it. The **import** statement specifies a class name including its package name. You can use the asterisk to import all classes in a package. If you use a class name without also providing an appropriate **import** statement, the Java compiler generates an error specifying that it does not recognize the class.

There is one package that is automatically imported for you: java.lang. This is the most basic of all packages since it defines fundamental classes like Object and String. You do not have to explicitly import a package when using these basic classes.

Iterating over the items in a collection

Use an Enumeration object with a **while** loop

```
import java.util.*;  // package with Enumeration
import com.apple.yellow.foundation.*;  // NSArray

double total = 0;
NSArray items = shoppingCart.items();
Enumeration e = items.objectEnumerator();
while (e.hasMoreElements()) {
   Product item = (Product)e.nextElement();
   total = total + item.price();
}
```

Don't modify the collection while enumerating

Iterating over the items in a collection

When using a collection, you will often need to process every object it contains. This is called iterating over the collection or enumerating the elements of a collection. Consider the **balance**() method of a shopping cart object: it must iterate over each of its items, get the price, and add it to the total. Java provides two tools for getting the job done—an Enumeration object and a **while** loop.

You get the enumeration object from the collection. It returns an object capable of enumerating all objects currently stored in that specific collection. Objects of type Enumeration respond to the following messages:

- **hasMoreElements**()—returns true if there are more objects to visit
- **nextElement**()—returns the next object in the collection

Use the enumeration object with a **while** loop to process each object in the collection. The while loop is a code block which is repeatedly executed as long as the conditional test—a boolean expression—evaluates to true:

```
while (condition) {
   loop body ...
}
```

While using an enumeration, you should not add and remove objects from the collection. Java also provides **for** and **do-until** loop statements not shown here.

Wrapper classes turn primitives into objects

Collections only store non-null object references

- Can't store *null* as a value in a collection
- Can't store primitive types—int, float, boolean, etc.

Java defines *wrapper classes* for treating primitives like objects

`Integer`	`Long`	`Float`	`Double`
`Short`	`Character`	`Byte`	`Boolean`

Required for some method arguments and return values in other classes

Wrapper classes are automatically imported—part of `java.lang`

Wrapper classes turn primitives into objects

Remember that Java is a hybrid language—not all data types are objects. Your code often makes use of simple values typed as int, double or boolean. In some cases, you need to treat these primitive values as objects. Collection classes like NSArray and NSDictionary cannot store primitive types. They can only store objects. Many other classes define method arguments and return values as object types and similarly, will not handle primitive types.

Java defines a special set of classes called wrapper classes. Their purpose is to wrap an object container around a primitive type. Wrapper classes enable you to turn primitive types into objects suitable for storing in a collection or passing to any method that requires a true object type. There is a specific wrapper class for each underlying primitive type—Integer for int, Double for double, and so on.

From a wrapper object, you can extract the original primitive type value. You can convert the type in both directions—from primitive to object and object back to primitive.

The wrapper classes are fundamental classes in the Java language. They are defined in the java.lang package which is automatically imported for you.

Conversions between primitive and object types

From primitive to object

```
int i = 10;
Integer number = new Integer(i);
```

From object to primitive

```
i = number.intValue();
```

Wrapper objects are immutable—you cannot modify the value

Conversions between primitive and object types

Here is a simple illustration. Assume you have a primitive type value, an int:

```
int i = 10;
```

You can create an instance of the Integer wrapper class that contains the int, thereby turning a primitive value into an object:

```
Integer number = new Integer(i);
```

You can now store this value in a collection such as an array:

```
array.addObject(number);
```

Later, you can retrieve the object from the collection and extract the original primitive value again:

```
Integer number = array.objectAtIndex(x);
int i = number.intValue();
```

The wrapper classes provide many additional capabilities for converting between different types, parsing values from strings, and generating values as formatted strings. Consult the Java documentation for additional details.

Additional foundation classes used with WebObjects

BigDecimal—arbitrary precision fixed point floating point number

```
import java.math.*;
```

NSGregorianDate—calendar date, time, and time zone

```
import com.apple.yellow.foundation.*;
```

NSData—buffer of arbitrary binary data

```
import com.apple.yellow.foundation.*;
```

Additional foundation classes used with WebObjects

WebObjects applications commonly make use of additional foundation classes. You should familiarize yourself with each of these classes.

The Java math package defines the BigDecimal class useful for representing large decimal numbers with specific rules for rounding and formatting. BigDecimal is ideal for storing monetary values. When you incorporate database connectivity into your WebObjects applications, you usually fetch number values as instances of BigDecimal.

The WebObjects foundation package defines the NSGregorianDate class. NSGregorianDates represent time and date values including a time zone. The class defines many convenience methods for comparing and calculating dates, and extracting pieces of the time and date like the month, the year, the minute, and the second. NSGregorianDate includes rich formatting capabilities and a simple way to ask for the current time.

The NSData class also comes from the WebObjects foundation package. It is used to represent an arbitrary buffer of binary data. Dynamically generated images or the contents of an uploaded file are good examples. Think of an NSData object as a collection of bytes that can be conveniently handled with a single object reference.

Behavior versus type—methods versus class

Objects can be of many different types—unrelated classes

```
ShoppingCart cart;
BankAccount account;
Inventory inventory;
```

But they can have analogous behavior—respond to the same messages

```
double balance = cart.balance();
double balance = card.balance();
double balance = inventory.balance();
```

Often, you need to type by behavior, not class

```
? thing = (?)items.objectAtIndex(0);
double balance = thing.balance();
```

Behavior versus type—methods versus class

In real-world Java programs, you use many objects of many diverse class types. They are often not related to each other in terms of the inheritance hierarchy. They don't share common superclasses except that they are all subclasses of the most generic class, Object. Consider how fundamentally different the following classes are from each other: ShoppingCart, BankAccount, Inventory.

Though unrelated in the class hierarchy, they have analogous behavior—they implement the same methods. What ShoppingCart, BankAccount, and Inventory might have in common is some aspect of their behavior: they each respond to the balance() method.

In many cases, you need to write code that works with a diverse set of objects that have common behavior, regardless of their dissimilar class types. You might write some code that takes any object and displays its balance to a user interface. You are depending on the fact that the object implements a balance() method. You specifically want to avoid making any assumptions about the class type of object. You don't really care about the class type at all.

This poses a simple coding problem: what type should the reference variable be? In this case, you want to type by behavior, not by class. A generic reference of type Object is not sufficient. The Object class does not define a balance() method.

Interfaces provide another kind of type

An *interface* defines a name for a group of related methods

An interface defines a type of behavior, independent from class type

A class is an implementation, an interface is only a specification

Often, you use an interface name rather than a class name

```
// objects with balance() behave like Assets
Asset thing = (Asset)items.objectAtIndex(0);
double balance = thing.balance();
```

Interfaces can also provide constants

```
double taxRate = Asset.TaxRate;
```

Packages define interfaces and classes implement them

Interfaces provide another kind of type

Java defines an alternate to classes for typing objects called interfaces. An interface provides a list of methods that define a type of behavior—a role. An interface has a name. An interface defines a formal type in Java.

Compare an interface with a class. A class defines all the attributes and all the methods valid for an object of that class. A class is a blueprint that describes the structure and origin an object. A class is an implementation. An interface is simply a list of methods. Interfaces typically capture only a subset of what an object can do. An interface is merely a specification that one or more classes may adhere to. A particular class may implement multiple interfaces.

Returning to the example, ShoppingCart, BankAccount, and Inventory are all class names. Each class definition provides details particular and specific to objects of that class. ShoppingCarts can add and remove products. BankAccounts have numbers and statements, credits and debits. An Inventory has a depreciation rate.

Yet ShoppingCart, BankAccount and Inventory objects all have a subset of behavior in common—they all respond to the balance() message. It is possible to define an interface called Asset that lists the messages you can send to any object that behaves like an Asset. In this case, the Asset interface defines one message—balance().

Use interfaces to type an object according to its behavior, regardless of its class.

Java naming conventions

Java is case-sensitive

`ShoppingCart` is different than `shoppingcart`

Class names are capitalized; inner words are capitalized

```
Asset
ShoppingCart
```

Method and variable names start with lowercase letters

```
index
checkOut()
```

Java defines many reserved words—use them only as intended

```
null, if, int, boolean . . .
```

Java naming conventions

Naming conventions are often just that: they are conventions rather than strictly enforced rules. But you should take them seriously. Faithfully adopting these rules will help you in several ways:

- You will be to understand another programmer's code more efficiently
- Another programmer will be able to understand your code more efficiently
- You can understand documentation more efficiently
- You are more likely to write correct code

Java is case-sensitive. "ShoppingCart" is different than "shoppingcart". Many compiler errors encountered by beginning programmers are merely misspellings that use the wrong case.

Class names are always capitalized. Multiple word names capitalize the first letter of each subsequent word. Method and variable names start with lowercase letters but also capitalize the first letter of subsequent words in multi-word names. Static constants vary in their naming, but usually begin with a capital letter. Often, the entire name is in uppercase.

Remember that there are many reserved names in Java—**null, while, if, int, boolean, void, null**—to list but a few. You should use them only as intended. Do not accidentally reuse them for your own variable names.

Common pitfalls

Compile-time errors

- Incorrect class or interface name—misspelled or missing import
- Incorrect method or attribute name—invalid or misspelled
- Missing a cast—method name is not valid for generic type

Runtime errors

- Sending a message to null—forgot to create and assign an object
- Applying an incorrect cast—the object is not the type you expect
- Exceeding the bounds of an array—index is negative or too high
- Adding a primitive or null to a collection—only non-null objects

Common pitfalls

If you are new Java programmer, you may likely run into a few common coding pitfalls. They show up as errors during compilation or at runtime. Often, the problem is simple: you forgot to import a class or you are missing a cast. One of the most common pitfalls shows up as a null pointer exception at runtime. The cause is typically that you forgot to create a new object and assign it to a reference variable. Without an object, you end up sending a message to null.

The Java compiler is quite good about providing meaningful error messages. Be sure to read the message carefully. The compiler typically lists the exact line number where the problem occurred. Review your code patiently until you locate the problem.

In many cases, the problem is not due to a fundamental lack of understanding but a peccadillo of coding syntax. With practice, you will learn to identify and remedy these common errors quickly. The list above provides some of the most common errors as a convenience to help you on your way.