CHAPTER 22

DungeonTrap Base Classes

Now that we have our game code set up and we can launch the game and get to the main menu screen, we can start building the various classes that define and power the DungeonTrap game. This game jam is a bit more complex than our first one, so I implore you to be patient and cautious with your coding. Take the time to check the completed project for this chapter to make sure your code is correct and to quickly grab the `import/using` statements for a given class.

Because this game is more complicated, there are some classes and class methods that are lengthy. I'll try my best to present the information as efficiently as possible. On your end though, you need to figure out the best strategy for learning what the code does as well as building out your copy of the class.

If following line by line and typing it up works for you, then be my guest; that's perfectly fine. However, you may find that copying and pasting in longer methods makes sense given the density of the code we're working with. As long as you understand what the code does, feel free to use either approach or a combination of the two. At the end of each major section of new code, I'll provide instructions on how to copy and paste the class files that were covered in that section into your project at the end of the chapter.

Before we jump into the deep end, let's take a look at the classes and enumerations we'll be working with. The enumerations that are used throughout the game to identify, organize, and compare different objects are listed in the following in the section labeled "TypeEnumerations." Take a look at the entries and let your imagination wander; try to think about how they might be used in the DungeonTrap game.

V. G. Brusca, *Introduction to Video Game Engine Development*, https://doi.org/10.1007/978-1-4842-7039-4_22

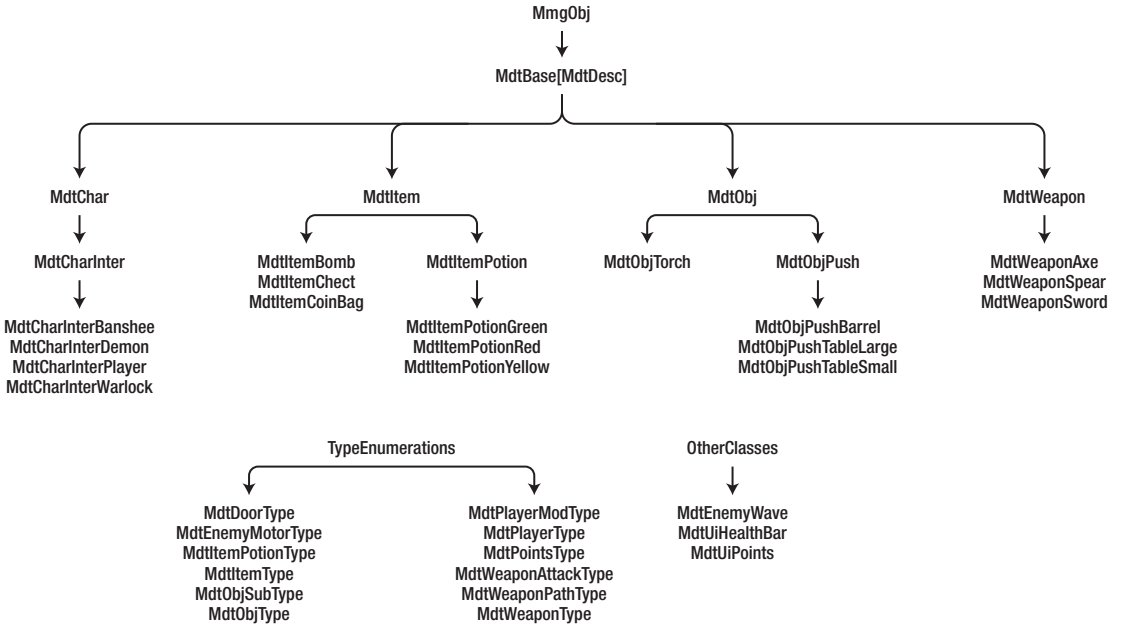


Figure 22-1. *MdtCharInter Class Inheritance Diagram 1*

The next aspect of the preceding class listing is the inheritance diagram of DungeonTrap's main classes. Similar to the technique of class extension we used in the game engine API, we've designed a class structure that uses base classes to encapsulate common functionality and present it to higher-level, more customized classes. Notice that the root of all game classes is the game engine's `MmgObj` class. Can you think of a reason why that is? One reason for this is that all the classes shown in the preceding diagram are drawn on the screen at one point or another. As such, they need to be able to plug into our game engine. By making the `MdtBase` class extend the `MmgObj` class, we ensure that any class that extends the `MdtBase` class can be processed by the game engine.

Take a moment to look over the diagram and try to imagine what the classes are like and how they might be used in our game. The remaining game classes that are not a part of the inheritance diagram are listed under the section labeled "OtherClasses." We'll be reviewing all the classes listed in the next few chapters, so don't worry if you're feeling overwhelmed. With that being said, we have our work cut out for us, so let's get to it!

DungeonTrap: Enumerations

The first group of classes we'll tackle are the enumerations. There are 12 of them, and they are all very simple and direct. We'll review them here. You can choose to type up the code as we cover each enumeration, or you can take advantage of a little shortcut and simply copy and paste the classes into your project. I'll provide instructions on how to do so at the end of this chapter.

Listing 22-1. MdtDoorType Class Code 1

```
public enum MdtDoorType {
    TOP_LEFT,
    TOP_RIGHT,
    LEFT,
    RIGHT,
    BOTTOM_LEFT,
    BOTTOM_RIGHT
}
```

Our first enumeration, `MdtDoorType`, is used to identify and differentiate between the six different doors on the DungeonTrap board. This enumeration is mainly used when prepping the level for a new wave of attackers.

Listing 22-2. MdtEnemyMotorType Class Code 1

```
public enum MdtEnemyMotorType {
    NONE,
    MOVE_X_THEN_Y,
    MOVE_Y_THEN_X
}
```

The `MdtEnemyMotorType` is an enumeration that is used to define the type of movement, motor, that is used by enemy players in the game.

Listing 22-3. MdtItemPotionType Class Code 1

```
public enum MdtItemPotionType {
    GREEN,
    YELLOW,
    RED,
    NONE
}
```

The `MdtItemPotionType` enumeration is used to indicate if an `MdtItemPotion` object is a green, red, or yellow potion. This enumeration is used by the `MdtItemPotion` base class and by extension the `MdtItemPotionGreen`, `MdtItemPotionRed`, and `MdtItemPotionYellow` classes.

Listing 22-4. MdtItemType Class Code 1

```
public enum MdtItemType {
    BOMB,
    CHEST,
    COIN_BAG,
    POTION_GREEN,
    POTION_RED,
    POTION_YELLOW
}
```

The `MdtItemType` enumeration is a more general enumeration used by the item base class `MdtItem`. By extension, the enumeration is a part of all item classes. You'll notice there is a bit of redundancy between this enumeration and the previous one. This is perfectly fine. Define your enumerations as you see fit when building your games.

Listing 22-5. MdtObjSubType Class Code 1

```
public enum MdtObjSubType {
    NONE,

    ITEM_BOMB,
    ITEM_CHEST,
    ITEM_COINS,
    ITEM_POTION,
```

```
    WEAPON_AXE,
    WEAPON_SPEAR,
    WEAPON_SWORD,

    MODIFIER_DOUBLE_POINTS,
    MODIFIER_PLAYER_INVINCIBLE,
    MODIFIER_ONE_SHOT_KILLS,

    ENEMY_BANSHEE,
    ENEMY_DEMON,
    ENEMY_WARLOCK,

    PLAYER_1,
    PLAYER_2,

    OBJECT_BARREL,
    OBJECT_TABLE_1,
    OBJECT_TABLE_2,
    OBJECT_TORCH,

    UI_HEALTH_BAR,
    UI_POINTS
}
```

The `MdtObjSubType` is an important enumeration that is used by all the classes that extend the `MdtBase` class. As you can see, there are entries for all the different types of objects in the game.

Listing 22-6. MdtObjType Class Code 1

```
public enum MdtObjType {
    NONE,
    ITEM,
    ENEMY,
    OBJECT,
    PLAYER,
    UI,
    WEAPON
}
```

Similar to the `MdtObjSubType`, the `MdtObjType` is used by just about every object in the game as well. The entries are used to describe objects at a higher level. The `MdtObjSubType` enumeration is used to provide more granular descriptions.

Listing 22-7. MdtPlayerModType Class Code 1

```
public enum MdtPlayerModType {
    NONE,
    INVINCIBLE,
    FULL_HEALTH,
    DOUBLE_POINTS,
    KILL_ALL
}
```

The `MdtPlayerModType` enumeration is responsible for marking different modifiers that can alter how a player's character performs.

Listing 22-8. MdtPlayerType Class Code 1

```
public enum MdtPlayerType {
    PLAYER_1,
    PLAYER_2,
    ENEMY,
    NONE
}
```

The next enumeration for us to review, listed in the preceding, is the `MdtPlayerType` enumeration; and it's used to differentiate between the different players of the game.

Listing 22-9. MdtPointsType Class Code 1

```
public enum MdtPointsType {
    PTS_100,
    PTS_250,
    PTS_500,
    PTS_1000,
}
```

The MdtPointsType enumeration is used to identify and differentiate between different point values. These are used throughout the game as part of the points system and are triggered by certain player actions.

Listing 22-10. MdtWeaponAttackType Class Code 1

```
public enum MdtWeaponAttackType {
    NONE,
    SLASHING,
    STABBING,
    THROWING,
}
```

The MdtWeaponAttackType enumeration is used to drive the behavior of the player's weapon. There are three types of attacks described by the enumeration, but not all of them have been implemented. I'll recommend some upgrades to the game that you can work on yourself at the end of this game build.

Listing 22-11. MdtWeaponPathType Class Code 1

```
public enum MdtWeaponPathType {
    NONE,
    PATH_1,
    PATH_2,
    PATH_3
}
```

The enumeration listed in the preceding is used to control the animation of a thrown weapon. There are three paths that the weapon can take that are defined by this enumeration.

Listing 22-12. MdtWeaponType Class Code 1

```
public enum MdtWeaponType {
    AXE,
    SPEAR,
    SWORD,
    WAND,
    NONE
}
```

The last enumeration we have to look at is the MdtWeaponType enumeration. It's used to define the behavior of the player's weapon. Again, not all weapons and items are actually used in the game. I've left some open for you to explore on your own. That brings us to the conclusion of this section. If you don't feel like typing up all the enumerations by hand, I placed a copy of all of them in a folder in the game engine's project directory.

Open the "cfg" folder and the "asset_src" folder contained inside. Find the "dungeon_trap_enum_classes" folder and open it. Copy all the files inside and paste them into the correct location, along with the other class files, in your DungeonTrap project. You may have to adjust the package name, for the Java version of the game, or the namespace, for the C# version of the game, of the newly pasted files to match your project setup. The classes are configured to work with the "game.jam.DungeonTrap" package or namespace.

DungeonTrap: Base Classes

In this section, we're going to continue on with building up the foundations of our game. The next few classes we'll review are the game's base classes, upon which all other classes are built. I'll try and explain the game specifications that are associated with the game classes we're building. The first snippet of code we'll look at is the definition for the MdtDesc interface. The interface definition is very concise, so we'll list it all right here.

Listing 22-13. MdtDesc Class Code 1

```
//Java Version
package game.jam.DungeonTrap;

import net.middlemind.MmgGameApiJava.MmgBase.MmgRect;

//C# Version
using System;
using net.middlemind.MmgGameApiCs.MmgBase;

namespace game.jam.DungeonTrap {

public interface MdtDesc {
    public MdtObjType GetMdtType();
    public void SetMdtType(MdtObjType obj);
```

```
    public MdtObjSubType GetMdtSubType();
    public void SetMdtSubType(MdtObjSubType obj);
    public MmgRect GetRect();
}
```

The MdtDesc interface is used only by the MdtBase class, and it defines the core set of information, in the form of method signatures that are available from any class that extends the MdtBase class. This means that we know the type, subtype, and display rectangle of any game object we encounter while building out the game. Let's take a look at the MdtBase class next. I will only use the strict review process that I've used previously in this text with classes that are larger and more complex. The MdtBase class happens to be very concise, so I'll just list it here. If I don't list C#-specific code and you're following along in C#, please check the completed chapter code.

Listing 22-14. MdtBase Class Code 1

```
public class MdtBase extends MmgObj implements MdtDesc {
    private MdtObjType mdtType = MdtObjType.NONE;
    private MdtObjSubType mdtSubType =  MdtObjSubType.NONE;

    01 public MdtBase() { }

    01 public MdtObjType GetMdtType() {
    02     return mdtType;
    03 }

    01 public void SetMdtType(MdtObjType obj) {
    02     mdtType = obj;
    03 }

    @Override
    01 public MdtObjSubType GetMdtSubType() {
    02     return mdtSubType;
    03 }

    @Override
    01 public void SetMdtSubType(MdtObjSubType obj) {
    02     mdtSubType = obj;
    03 }
```

```
    @Override
    01 public MmgRect GetRect() {
    02    return new MmgRect(GetPosition(), GetWidth(), GetHeight());
    03 }
}
```

Notice that the `MdtBase` class implements the `MdtDesc` interface while extending the `MmgObj` class. As such, the `MdtBase` class defines the fields necessary to support the required class methods. The `mdtType` and `mdtSubType` fields drive the get and set methods `GetMdtType`, `SetMdtType`, `GetMdtSubType`, and `SetMdtSubType` listed in the preceding.

Take a look at the `GetRect` method signature. Note that we're using game engine API classes. Don't forget that the game we're building is an extension of the game engine API. There's one very important part of the class outline listed in the preceding. Can you see it? It's the two words "extends MmgObj" that are part of the class definition. What this means is that the `MdtBase` class extends the functionality provided by the `MmgObj` class. This lets any `MdtBase` class instance be processed by the game engine because any `MdtBase` object is also an `MmgObj` object.

The next few classes that we'll review in this chapter are the first-level base classes that power all the other game classes. They are the `MdtChar`, `MdtItem`, `MdtObj`, and `MdtWeapon` classes. These classes indirectly address the game specifications that mention the player fighting enemies and getting items. We actually built into the game the ability for the player to push some objects sending them flying into enemies or just the wall.

Because these are complex classes that encapsulate a lot of functionality, we'll employ the more rigid review structure that we used in Parts 1 and 2 of this text. We'll review these classes using the approach listed in the following. Only the sections that apply to any given class will be used in the class' review:

1. Static class members
2. Enumerations
3. Class fields
4. Pertinent method outline
5. Support method details
6. Main method details

Let's start with the MdtChar class. You can choose to type up your version of the class as we review it or on your own while looking at this chapter's completed project code. Lastly, you can opt to copy the code in one step. I'll provide instructions on how to do so at the end of this chapter. Just make sure you understand what the code is doing before you add it to your game. The MdtChar class doesn't have any static class members or enumerations to review, so we'll start things off with the class' fields. Let's look at some code.

MdtChar: Class Review

The MdtChar class is the base class that powers players and enemy characters in the DungeonTrap game. Pay attention to the capabilities that are supported by the MdtChar class because we'll see them again in the player and enemy classes that we'll soon be working on.

MdtChar: Class Fields

Let's take a look at the class' fields. I'll try to keep the fields grouped logically when applicable.

Listing 22-15. MdtChar Class Fields 1

```
public MmgSprite subj;
public int healthCurrent = 3;
public int healthMax = 3;
public MdtPlayerType healthDamagedBy = MdtPlayerType.NONE;
public boolean lret = false;
public int speed = ScreenGame.GetSpeedPerFrame(60);
public int dir = MmgDir.DIR_NONE;
```

The first group of class fields for us to look at has various fields that help define the character, how it looks on the screen, and the character's movement among other things. The subj field is an instance of the MmgSprite class. Recall from Part 1 of this text that the MmgSprite class is used whenever a game object has a series of frames used to animate the object.

The subsequent two fields, `healthCurrent` and `healthMax`, are used to track the character's health and damage. The `healthDamagedBy` field is used to track the type of character that damaged this character. The `lret` field is used internally by certain class methods as a temporary variable. The remaining two fields, `speed` and `dir`, define the character's speed and indicate the character's current direction, respectively.

Listing 22-16. MdtChar Class Fields 2

```
public int frameFrontStart = 0;
public int frameFrontStop = 0;
public int frameBackStart = 0;
public int frameBackStop = 0;
public int frameLeftStart = 0;
public int frameLeftStop = 0;
public int frameRightStart = 0;
public int frameRightStop = 0;
```

The next block of class fields listed in the preceding are used to control the sprite subject's animation frames. As you can see, there are start and stop fields for the four cardinal directions. The start and stop fields represent the indexes of the animation frames to use for the given direction. This is how we create walking animations that are separate for each direction.

Listing 22-17. MdtChar Class Fields 3

```
public MmgRect current = null;
public boolean isMoving = false;
public boolean isAttacking = false;
private Random rand = null;
public MdtWeapon weaponCurrent = null;
public Hashtable<String, MdtWeapon> weapons = null;
public ScreenGame screen = null;
public MdtBase coll = null;
```

The last block of fields wraps up the class field review for the `MdtChar` class. The current field is used internally in determining if the character can move in the current direction. The `isMoving` and `isAttacking` Boolean flags are used to indicate the character's current action. The next class field, `rand`, is used to generate random values when needed by certain class methods.

The `weaponCurrent` field tracks which weapon is currently active for the current player. The weapons data structure that follows the `weaponCurrent` field is representative of the character's current set of weapons. The screen field is a reference to the game's main game screen class and is used to call methods on the screen class. The last entry is the `coll` field. This field is used internally by certain methods as part of the character collision detection process.

That concludes the class' field review. Up next, we'll take a look at the class' method outline. Make sure to carefully add these class fields to your copy of the `MdtChar` class.

MdtChar: Pertinent Method Outline

I've organized the class methods into the three groups listed in the following. Take a quick look at them. We'll review the methods in more detail in upcoming class review sections.

Listing 22-18. MdtChar Pertinent Method Outline 1

```
//Main Methods
public MdtChar() { ... }
public MdtChar(MmgSprite Subj, int fsFront, int feFront, int fsLeft, int
feLeft, int fsRight, int feRight, int fsBack, int feBack, ScreenGame
Screen, MdtObjType ObjType, MdtObjSubType ObjSubType) { ... }

public void TakeDamage(int i, MdtPlayerType p) { ... }

public void SetHealthToMax() { ... }

//Support Methods
public MmgSprite GetSubj() { ... }
public void SetSubj(MmgSprite Subj) { ... }
public int GetHealthCurrent() { ... }
public void SetHealthCurrent(int curr) { ... }
```

```
public int GetHealthMax() { ... }
public void SetHealthMax(int hMax) { ... }
public MdtPlayerType GetHealthDamagedBy() { ... }
public void SetHealthDamagedBy(MdtPlayerType p) { ... }

public int GetSpeed() { ... }
public void SetSpeed(int s) { ... }
public int GetDir() { ... }
public void SetDir(int Dir) { ... }
public MmgRect GetCurrentCollRect() { ... }
public void SetCurrentCollRect(MmgRect curr) { ... }
public boolean GetIsMoving() { ... }
public void SetIsMoving(boolean b) { ... }
public boolean GetIsAttacking() { ... }
public void SetIsAttacking(boolean b) { ... }
public Random GetRand() { ... }
public void SetRand(Random r) { ... }
public MdtWeapon GetWeaponCurrent() { ... }
public void SetWeaponCurrent(MdtWeapon weapon) { ... }

public Hashtable<String, MdtWeapon> GetWeapons() { ... }

public void SetWeapons(Hashtable<String, MdtWeapon> wps) { ... }

public ScreenGame GetScreen() { ... }
public void SetScreen(ScreenGame o) { ... }
public MdtBase GetCurrentColl() { ... }
public void SetCurrentColl(MdtBase c) { ... }

//Support Methods Animation Frames
public int GetFrameFrontStart() { ... }
public void SetFrameFrontStart(int frame) { ... }
public int GetFrameFrontStop() { ... }
public void SetFrameFrontStop(int frame) { ... }

public int GetFrameBackStart() { ... }
public void SetFrameBackStart(int frame) { ... }
public int GetFrameBackStop() { ... }
public void SetFrameBackStop(int frame) { ... }
```

```
public int GetFrameLeftStart() { ... }
public void SetFrameLeftStart(int frame) { ... }
public int GetFrameLeftStop() { ... }
public void SetFrameLeftStop(int frame) { ... }

public int GetFrameRightStart() { ... }
public void SetFrameRightStart(int frame) { ... }
public int GetFrameRightStop() { ... }
public void SetFrameRightStop(int frame) { ... }
```

That concludes the `MdtChar` class' method outline. Take a moment to look over the methods and imagine how the class functions as a whole. I wanted to also list the class definitions in this section.

Listing 22-19. MdtChar Class Definitions 1

```
//Java Version
public class MdtChar extends MdtBase {

//C# Version
public class MdtChar : MdtBase {
```

As you can see from the lines listed in the preceding, the syntax for the class definition in Java is very similar to the syntax used in C#. If you have any questions, take a look at the completed chapter code in the game engine's project folder. In the next sections, we'll review the class' methods in detail.

MdtChar: Support Method Details

The `MdtChar` class' support methods are all basic get and set methods associated with the class fields we just finished reviewing. Due to their simplicity, I'll list the methods here, but we won't go into any detail regarding them. Be sure to read over the methods and make sure you understand them before adding them to your version of the `MdtChar` class.

Listing 22-20. MdtChar Support Method Details 1

```
//Support Methods
01 public MmgSprite GetSubj() {
02     return subj;
03 }

01 public void SetSubj(MmgSprite Subj) {
02     subj = Subj;
03 }

01 public int GetHealthCurrent() {
02     return healthCurrent;
03 }

01 public void SetHealthCurrent(int curr) {
02     healthCurrent = curr;
03 }

01 public int GetHealthMax() {
02     return healthMax;
03 }

01 public void SetHealthMax(int hMax) {
02     healthMax = hMax;
03 }

01 public MdtPlayerType GetHealthDamagedBy() {
02     return healthDamagedBy;
03 }

01 public void SetHealthDamagedBy(MdtPlayerType p) {
02     healthDamagedBy = p;
03 }

01 public int GetSpeed() {
02     return speed;
03 }
```

```
01 public void SetSpeed(int s) {
02     speed = s;
03 }

01 public int GetDir() {
02     return dir;
03 }

01 public void SetDir(int Dir) {
02     dir = Dir;
03 }

01 public MmgRect GetCurrentCollRect() {
02     return current;
03 }

01 public void SetCurrentCollRect(MmgRect curr) {
02     current = curr;
03 }

01 public boolean GetIsMoving() {
02     return isMoving;
03 }

01 public void SetIsMoving(boolean b) {
02     isMoving = b;
03 }

01 public boolean GetIsAttacking() {
02     return isAttacking;
03 }

01 public void SetIsAttacking(boolean b) {
02     isAttacking = b;
03 }

01 public Random GetRand() {
02     return rand;
03 }
```

```
01 public void SetRand(Random r) {
02     rand = r;
03 }

01 public MdtWeapon GetWeaponCurrent() {
02     return weaponCurrent;
03 }

01 public void SetWeaponCurrent(MdtWeapon weapon) {
02     weaponCurrent = weapon;
03 }

01 public Hashtable<String, MdtWeapon> GetWeapons() {
02     return weapons;
03 }

01 public void SetWeapons(Hashtable<String, MdtWeapon> wps) {
02     weapons = wps;
03 }

01 public ScreenGame GetScreen() {
02     return screen;
03 }

01 public void SetScreen(ScreenGame o) {
02     screen = o;
03 }

01 public MdtBase GetCurrentColl() {
02     return coll;
03 }

01 public void SetCurrentColl(MdtBase c) {
02     coll = c;
03 }
```

In keeping with the grouping used by the "MdtChar: Class Fields" section, I've listed the animation frame support methods in the following group.

Listing 22-21. MdtChar Support Method Details 2

```
//Support Methods Animation Frames
01 public int GetFrameFrontStart() {
02     return frameFrontStart;
03 }

01 public void SetFrameFrontStart(int frame) {
02     frameFrontStart = frame;
03 }

01 public int GetFrameFrontStop() {
02     return frameFrontStop;
03 }

01 public void SetFrameFrontStop(int frame) {
02     frameFrontStop = frame;
03 }

01 public int GetFrameBackStart() {
02     return frameBackStart;
03 }

01 public void SetFrameBackStart(int frame) {
02    frameBackStart = frame;
03 }

01 public int GetFrameBackStop() {
02     return frameBackStop;
03 }

01 public void SetFrameBackStop(int frame) {
02     frameBackStop = frame;
03 }

01 public int GetFrameLeftStart() {
02     return frameLeftStart;
03 }
```

```
01 public void SetFrameLeftStart(int frame) {
02     frameLeftStart = frame;
03 }

01 public int GetFrameLeftStop() {
02     return frameLeftStop;
03 }

01 public void SetFrameLeftStop(int frame) {
02     frameLeftStop = frame;
03 }

01 public int GetFrameRightStart() {
02     return frameRightStart;
03 }

01 public void SetFrameRightStart(int frame) {
02     frameRightStart = frame;
03 }

01 public int GetFrameRightStop() {
02     return frameRightStop;
03 }

01 public void SetFrameRightStop(int frame) {
02     frameRightStop = frame;
03 }
```

If you're following along in C#, then please take a moment to review this class in the completed project code for this chapter. There are some slight differences in the data structure used to keep track of the character's weapons. In C#, we use a `Dictionary` as opposed to the `Hashtable` class used in the Java version. In the next section, we'll take a look at the class' main methods.

MdtChar: Main Method Details

In this section, we'll take a quick look at the class' main methods.

Listing 22-22. MdtChar Main Method Details 1

```
01 public MdtChar() {
02
03 }
```

```
01 public MdtChar(MmgSprite Subj, int fsFront, int feFront, int fsLeft, int
   feLeft, int fsRight, int feRight, int fsBack, int feBack, ScreenGame
   Screen, MdtObjType ObjType, MdtObjSubType ObjSubType) {
02     SetSubj(Subj);
03
04     SetFrameFrontStart(fsFront);
05     SetFrameFrontStop(feFront);
06     SetFrameLeftStart(fsLeft);
07     SetFrameLeftStop(feLeft);
08     SetFrameRightStart(fsRight);
09     SetFrameRightStop(feRight);
10     SetFrameBackStart(fsBack);
11     SetFrameBackStop(feBack);
12
13     SetRand(new Random(System.currentTimeMillis()));
14     SetScreen(Screen);
15     SetMdtType(ObjType);
16     SetMdtSubType(ObjSubType);
17
18     SetHeight(subj.GetHeight());
19     SetWidth(subj.GetWidth());
20 }
```

```
01 public void TakeDamage(int i, MdtPlayerType p) {
02     SetHealthDamagedBy(p);
03     healthCurrent -= i;
04     if(healthCurrent < 0) {
05         healthCurrent = 0;
06     }
07 }
```

```
01 public void SetHealthToMax() {
02     healthCurrent = healthMax;
03 }
```

The main methods listed in the preceding consist of a pair of class constructors followed by a set of methods that are used to alter the character's current health indicating that the character has taken damage. The first constructor listed is a generic constructor that takes no arguments. This constructor places the responsibility of configuring the class on the developer.

The second class constructor listed in the preceding takes a full set of arguments used to configure the class and assign values to all pertinent class fields. Notice the use of the `ScreenGame`, `MdtObjType`, and `MdtObjSubType` objects as constructor arguments. The second to last main method listed in the preceding is the `TakeDamage` method. This method is used to update the current character's damage. On lines 3–5, if the damage is lower than zero, the amount is corrected and set to zero. The last method shown restores the current character back to full health.

That concludes our review of the `MdtChar` class. We won't be covering a demonstration of the base classes reviewed in this chapter. We'll see these classes in action when we review the level 1 and 2 extended classes. We'll also see them in action when we build out the game's main game screen and all the included game functionality. Next up, we'll take a look at the base class that drives the behavior of items in the game.

MdtItem: Class Review

The `MdtItem` class is the base class upon which all game items are extended. Because the nature of game items is almost identical, we can encapsulate a lot of functionality in the base class that we know will be used, and not redefined, by the item classes that extend it. The `MdtItem` class has no static class members or pertinent enumerations to speak of, so we'll skip those sections and begin with the review of the class' fields.

MdtItem: Class Fields

Let's take a look at the `MdtItem` class' fields. There aren't too many, so I've listed them in the following group.

Listing 22-23. MdtItem Class Fields 1

```
public MmgObj subj = null;
private boolean lret = false;
public MdtPointsType points = MdtPointsType.PTS_100;
public boolean canVanish = true;
public long displayTime = 0;
public long displayTimeTotal = 3000;
public ScreenGame screen = null;
```

The first entry in the block of class fields is the `subj` field. Notice that this is a generic `MmgObj` instance. This means that we can use an `MmgSprite` or an `MmgBmp` object among others. We can use any drawable class that extends the `MmgObj` class. This gives us a lot of flexibility as to how we can implement an `MdtItem`. The following field, `lret`, is a private field used in certain class methods. The `points` field indicates how much the item is worth. The points are awarded to the player that picks up the item.

The next three class fields, `canVanish`, `displayTime`, and `displayTimeTotal`, are all used to activate and implement the `MdtItem` class' vanish functionality. If `canVanish` is set to true, then the class will measure how long the item has been visible, `displayTime`, and remove it from the game once the value of `displayTime` is greater than the value of the `displayTimeTotal` field. The last entry listed in the preceding is the `screen` field. This field holds a reference to the game's main game screen.

It's always useful to be able to call methods on the main game screen, so we keep a reference around. You'll see this pattern again in many other classes. That concludes the class' field review. Up next, we'll take a look at the class' method outline. Make sure to carefully add these fields to your copy of the `MdtItem` class.

MdtItem: Pertinent Method Outline

The `MdtItem` class' method outline is listed in the following with methods grouped into two main categories, main and support methods.

Listing 22-24. MdtItem Pertinent Method Outline 1

```
//Main Methods
public MdtItem() { ... }
public MdtItem(MmgObj Subj, MdtObjType ObjType, MdtObjSubType ObjSubType,
MdtPointsType Points) { ... }
```

```
public void SetPosition(MmgVector2 v) { ... }
public void SetPosition(int x, int y) { ... }
public void SetX(int i) { ... }
public void SetY(int i) { ... }
public boolean MmgUpdate(int updateTick, long currentTimeMs, long
msSinceLastFrame) { ... }

public void MmgDraw(MmgPen p) { ... }

//Support Methods
public MmgObj GetSubj() { ... }
public void SetSubj(MmgObj Subj) { ... }
public MdtPointsType GetPoints() { ... }
public void SetPoints(MdtPointsType i) { ... }
public boolean GetCanVanish() { ... }
public void SetCanVanish(boolean b) { ... }
public long GetDisplayTime() { ... }
public void SetDisplayTime(long l) { ... }
public long GetDisplayTimeTotal() { ... }
public void SetDisplayTimeTotal(long l) { ... }
public ScreenGame GetScreen() { ... }
public void SetScreen(ScreenGame o) { ... }
```

That brings us to the end of the `MdtItem` class' method outline. I wanted to list the class definitions here for both Java and C# versions of the class. Pay special attention to the `super/base` class that the `MdtItem` class extends.

Listing 22-25. MdtItem Class Definition 1

```
//Java Version
public class MdtItem extends MdtBase {

//C# Version
public class MdtItem : MdtBase {
```

In the next section, we'll review the class' support methods in detail.

MdtItem: Support Method Details

The `MdtItem` class' support methods are all basic get and set methods for their associated class fields. Because these methods are very simple, I'll just list them here. We won't go into any detail regarding them. Be sure to read over the methods and understand how they work before adding them into your version of the `MdtItem` class.

Listing 22-26. MdtItem Support Method Details 1

```
01 public MmgObj GetSubj() {
02     return subj;
03 }

01 public void SetSubj(MmgObj Subj) {
02     subj = Subj;
03 }

01 public MdtPointsType GetPoints() {
02     return points;
03 }

01 public void SetPoints(MdtPointsType i) {
02     points = i;
03 }

01 public boolean GetCanVanish() {
02     return canVanish;
03 }

01 public void SetCanVanish(boolean b) {
02     canVanish = b;
03 }

01 public long GetDisplayTime() {
02     return displayTime;
03 }

01 public void SetDisplayTime(long l) {
02     displayTime = l;
03 }
```

```
01 public long GetDisplayTimeTotal() {
02     return displayTimeTotal;
03 }

01 public void SetDisplayTimeTotal(long l) {
02     displayTimeTotal = l;
03 }

01 public ScreenGame GetScreen() {
02     return screen;
03 }

01 public void SetScreen(ScreenGame o) {
02     screen = o;
03 }
```

That concludes the review of the class' support methods. In the next section, we'll list and review the `MdtItem` class' main methods.

MdtItem: Main Method Details

In this class review section, we'll take a look at the class' main methods. We'll cover these methods in more detail than the supporting methods. As always, you can type up the code by hand from the text or from the completed chapter project's classes. You can always choose to copy and paste the completed version of the class into your project, and I'll provide instructions on how to do so at the end of this chapter.

Listing 22-27. MdtItem Main Method Details 1

```
01 public MdtItem() {
02
03 }

01 public MdtItem(MmgObj Subj, MdtObjType ObjType, MdtObjSubType
   ObjSubType, MdtPointsType Points) {
02     SetSubj(Subj);
03     SetMdtType(ObjType);
04     SetMdtSubType(ObjSubType);
```

```
05      SetPoints(Points);
06      SetWidth(subj.GetWidth());
07      SetHeight(subj.GetHeight());
08      SetDisplayTimeTotal((MmgHelper.GetRandomIntRange(3, 8) * 1000));
09  }
```

```
@Override
01  public boolean MmgUpdate(int updateTick, long currentTimeMs, long
    msSinceLastFrame) {
02      lret = false;
03      if (isVisible == true) {
04          subj.MmgUpdate(updateTick, currentTimeMs, msSinceLastFrame);
05          if(screen != null && canVanish) {
06              displayTime += msSinceLastFrame;
07              if(displayTime >= displayTimeTotal) {
08                  screen.RemoveObj(this);
09              }
10          }
11          lret = true;
12      }
13      return lret;
14  }
```

```
@Override
01  public void MmgDraw(MmgPen p) {
02      if (isVisible == true) {
03          subj.MmgDraw(p);
04      }
05  }
```

The first block of main methods for us to review are listed in the preceding. The first entry is a basic class constructor that takes no arguments and performs no class configuration. If you use this constructor, you'll need to configure the class yourself. The next main method listed is an advanced constructor that takes arguments used to set the values of pertinent class fields. Notice that on line 8 of this constructor, the total display time is set to a random number of seconds, between 3 and 8 seconds.

The next two main methods listed in this block are the MmgUpdate and MmgDraw methods that are used by the game engine's drawing routine. On line 4, we call the MmgUpdate method of the subj field to ensure any necessary updates are completed. On lines 5–10, the vanish functionality of the MdtItem class is implemented. Note that this feature is only enabled if the canVanish class field is set to true and the screen field is not null. The code on line 8 is used to remove the current item from the game if it has vanished.

The last main method in this block is the MmgDraw method. Notice how simple this method is. Its main purpose is to ensure the subj field is drawn to the screen if the isVisible flag is set to true. The next block of main methods, listed in the following, override the default MmgObj methods provided through the MdtBase class. The overridden version of these methods is designed to update the subj field along with changes to the MdtItem class itself.

Listing 22-28. MdtItem Main Method Details 2

```
@Override
01 public void SetPosition(MmgVector2 v) {
02    super.SetPosition(v);
03    subj.SetPosition(v);
04 }

@Override
01 public void SetPosition(int x, int y) {
02     super.SetPosition(x, y);
03     subj.SetPosition(x, y);
04 }

@Override
01 public void SetX(int i) {
02     super.SetX(i);
03     subj.SetX(i);
04 }

@Override
01 public void SetY(int i) {
02     super.SetY(i);
03     subj.SetY(i);
04 }
```

As you can see from the methods listed in the preceding, the super class, or base class in C#, is being kept synchronized with the `subj` field. This is a common setup for DungeonTrap game classes that you'll see time and time again. If you're following along in C#, be aware you have to adjust how the method override is declared. Check the completed chapter project if there is any confusion.

That concludes our review of the `MdtItem` class. Up next, we'll tackle a similar class that powers the game's non-item objects like tables and barrels. We're going to continue moving through the game's base classes and extended classes in this and the next chapter. At that point, we'll check the DungeonTrap game's specifications, in Chapter 15, to get an idea of where we are in the game development process.

MdtObj: Class Review

The `MdtObj` class is similar to the `MdtItem` class, but it's simpler in most respects. In the DungeonTrap game, there are objects that interact with the player but not like items. Some `MdtObj` instances are for atmosphere as in the case of the flickering torches. Other instances are interactive game objects like the barrels and tables that randomly appear after each wave of enemies.

In the game, you can push barrels and tables to deal damage to enemy characters. You'll encounter those more advanced object classes soon. They all extend the `MdtObj` class. Let's take a look at some code!

MdtObj: Class Fields

The `MdtObj` class has only two class fields. The first is the `subj` field, an instance of the `MmgObj` class. The second field is a private Boolean flag.

Listing 22-29. MdtObj Class Fields 1

```
public MmgObj subj = null;
private boolean lret = false;
```

Notice that the `subj` field is an instance of the very general `MmgObj` class. This gives us a lot of flexibility as to how we can display the game object. The `lret` field is an internal variable used by certain class methods.

MdtObj: Pertinent Method Outline

The MdtObj class methods are listed in the following in two groups, main methods and supports methods, respectively.

Listing 22-30. MdtObj Pertinent Method Outline 1

```
//Main Methods
public MdtObj() { ... }
public MdtObj(MmgObj Subj, MdtObjType ObjType, MdtObjSubType ObjSubType) {
... }

public void SetPosition(MmgVector2 v) { ... }
public void SetPosition(int x, int y) { ... }
public void SetX(int i) { ... }
public void SetY(int i) { ... }
public boolean MmgUpdate(int updateTick, long currentTimeMs, long
msSinceLastFrame) { ... }

public void MmgDraw(MmgPen p) { ... }

//Support Methods
public MmgObj GetSubj() { ... }
public void SetSubj(MmgObj Subj) { ... }
```

If you're following along in C#, pay special attention to how the method override is handled. There is a slight syntax change between the two languages. Similarly, note the syntax difference in the following class definitions.

Listing 22-31. MdtObj Class Definitions 1

```
//Java Version
public class MdtObj extends MdtBase {

//C# Version
public class MdtObj : MdtBase {
```

That concludes the MdtObj class method outline and class definition review. Next, we'll look at the MdtObj class' support methods.

MdtObj: Support Method Details

There are only two support methods in the `MdtObj` class: the get and set methods `GetSubj` and `SetSubj`. I'll list the two support methods here; then we'll move on to the next review section.

Listing 22-32. MdtObj Support Method Details 1

```
01 public MmgObj GetSubj() {
02     return subj;
03 }

01 public void SetSubj(MmgObj Subj) {
02     subj = Subj;
03 }
```

MdtObj: Main Method Details

The first block of class constructors for us to review are listed in the following. Take a quick look at them before we review them and dive into the remaining main methods.

Listing 22-33. MdtObj Main Method Details 1

```
01 public MdtObj() {
02
03 }

01 public MdtObj(MmgObj Subj, MdtObjType ObjType, MdtObjSubType ObjSubType) {
02     SetSubj(Subj);
03     SetMdtType(ObjType);
04     SetMdtSubType(ObjSubType);
05     SetWidth(subj.GetWidth());
06     SetHeight(subj.GetHeight());
07 }
```

The first entry is a basic class constructor that takes no arguments. The second entry is an advanced constructor that takes arguments used to set the values of pertinent class fields. The next block of methods for us to review are the position method overrides.

Listing 22-34. MdtObj Main Method Details 2

```
@Override
01 public void SetPosition(MmgVector2 v) {
02     super.SetPosition(v);
03     subj.SetPosition(v);
04 }

@Override
01 public void SetPosition(int x, int y) {
02     super.SetPosition(x, y);
03     subj.SetPosition(x, y);
04 }

@Override
01 public void SetX(int i) {
02     super.SetX(i);
03     subj.SetX(i);
04 }

@Override
01 public void SetY(int i) {
02     super.SetY(i);
03     subj.SetY(i);
04 }
```

The position method overrides that are listed in the preceding are very similar to other versions of these methods we've seen before. This pattern will come up again in other `MdtObj` classes. The last two methods for us to review are the `MmgUpdate` and `MmgDraw` methods.

Listing 22-35. MdtObj Main Method Details 3

```
@Override
01 public boolean MmgUpdate(int updateTick, long currentTimeMs, long
   msSinceLastFrame) {
02     lret = false;
03     if (isVisible == true) {
```

```
04          subj.MmgUpdate(updateTick, currentTimeMs, msSinceLastFrame);
05          lret = true;
06      }
07      return lret;
08  }
```

```
@Override
01 public void MmgDraw(MmgPen p) {
02      if (isVisible == true) {
03          subj.MmgDraw(p);
04      }
05  }
```

The first entry listed in the preceding is the `MmgUpdate` method. Notice how the method is calling the `MmgUpdate` method of the `subj` field. This connects the `subj` object to the game engine's drawing routine. The next method listed, `MmgDraw`, is responsible for rendering this class to the screen. Both the `MdtItem` and `MdtObj` classes are objects drawn on the game board that will interact with the game's players via collisions. That concludes this review of the `MdtObj` class' main methods. Up next, we'll take a look at the `MdtWeapon` base class.

MdtWeapon: Class Review

The `MdtWeapon` class is the base class of all supported weapons. While there are three weapon classes implemented, the game only supports one weapon, the spear. I'll talk about some of the possibilities for expanding and personalizing at the end of this game build. The `MdtWeapon` class does not have any static class members or enumerations to cover, so we'll start things off with a review of the class' fields.

MdtWeapon: Class Fields

The `MdtWeapon` class has a number of class fields that support attacking with a stabbing- or throwing-style weapon. Let's have a look.

Listing 22-36. MdtWeapon Class Fields 1

```
public MmgBmp subjFront = null;
public MmgBmp subjBack = null;
public MmgBmp subjLeft = null;
public MmgBmp subjRight = null;
public MdtChar holder = null;
public boolean active = false;
private boolean lret = false;
```

The first block of class fields for us to review starts with four subject fields. The `subjFront`, `subjBack`, `subjLeft`, and `subjRight` class fields are used to draw the weapon in each direction the character holding the weapon can face. The next field listed, `holder`, is an instance of the `MdtChar` class and represents the character, player or enemy, holding the weapon. The `active` field is a Boolean flag used to indicate the active weapon in a list of weapons held by a character. The last field listed, `lret`, is a Boolean flag used internally by certain class methods.

Listing 22-37. MdtWeapon Class Fields 2

```
public MdtWeaponType weaponType = MdtWeaponType.NONE;
public long animTimeMsCurrent = 0;
public long animTimeMsTotal = 500;
public double animPrctComplete = 0.0d;
public int damage = 1;
private MmgRect src = null;
private MmgRect dst = null;
```

The next block of class fields describe aspects of the weapon's behavior and help render the weapon. The `weaponType` field is used to describe the type of weapon. The `animTimeMsCurrent` field tracks the current time in milliseconds the weapon animation has progressed. The `animTimeMsTotal` field represents the total time the animation has to run. The `animPrctComplete` field tracks how far along the animation has progressed, expressed as a percentage.

The following class field, `damage`, indicates how much damage this weapon does when it hits an enemy character. Finally, the last two fields listed in the preceding, `src` and `dst`, are used in the render process for stabbing weapons. The next block of class fields holds information necessary to render the throwing version of the weapon.

Listing 22-38. MdtWeapon Class Fields 3

```
public int throwingSpeed = ScreenGame.GetSpeedPerFrame(120);
public int throwingSpeedSkew = ScreenGame.GetSpeedPerFrame(40);
public int throwingFrame = 0;
public int throwingDir = 0;
public long throwingCoolDown = 500;
public long throwingTimeMsCurrent = 0;
public long throwingTimeMsRotation = 200;
```

The block of class fields listed in the preceding are used in the animation process for the thrown weapon. The `throwingSpeed` field controls how fast the thrown weapon travels. The subsequent field determines how much skew, off of a straight throw, the weapon has. This adds a slight angle to some of the weapon throws. The `throwingFrame` field is used to track the current frame used for the thrown weapon's animations. The `throwingDir` field sets the direction the weapon is to be thrown.

The next field listed in the preceding, `throwingCoolDown`, determines how much time must pass before the next use of the weapon. The last two class fields are timing values. The `throwingTimeMsCurrent` field is used to track the current time, in the thrown weapon's animation, that has progressed. The last field listed in the preceding is used in the thrown weapon's animation to determine how fast the projectile rotates. That brings us to the last block of class fields to review.

Listing 22-39. MdtWeapon Class Fields 4

```
public long stabbingCoolDown = 150;
public ScreenGame screen = null;
private int tmpI = 0;
public MmgRect current = null;
public MdtPlayerType player;
public MdtBase coll = null;
public MdtWeaponAttackType attackType = MdtWeaponAttackType.NONE;
public MdtWeaponPathType throwingPath = MdtWeaponPathType.NONE;
private Random rand = null;
```

The `stabbingCoolDown` field is similar to its throwing counterpart in that it determines how much time has to pass before a new stab attack can be performed. The `screen` class field references the game's main game screen and is used to detect

collisions with other game objects, among other things. The field `tmpI` is used internally by certain class methods. Subsequently, the `current` field is used to hold the weapon's current position and dimension. Next up is the player field, which is used to indicate what type of player is holding the weapon.

The `coll` field is used in the weapon's collision detection, while the `attackType` field indicates which attack animation to use for the current weapon. The `throwingPath` class field is used to separate the paths a thrown weapon can take. Lastly, the `rand` field is used to generate random values used by this class. That brings us to the end of the class field review. Add these fields to your copy of the `MdtWeapon` class if you haven't already. You can type them in by hand, or you can copy them from the chapter's completed project code. The choice is yours.

MdtWeapon: Pertinent Method Outline

In this section, we'll list the MdtWeapon class' methods in two groups, main methods and supports methods.

Listing 22-40. MdtWeapon Pertinent Method Outline 1

```
//Main Methods
public MdtWeapon(MdtChar Holder, MdtWeaponType WeaponType, MdtPlayerType
Player) { ... }

public MdtWeapon(MdtChar Holder, MdtWeaponType WeaponType,
MdtWeaponAttackType AttackType, MdtPlayerType Player) { ... }

public MdtWeapon Clone() { ... }
public boolean MmgUpdate(int updateTick, long currentTimeMs, long
msSinceLastFrame) { ... }

public MmgRect GetWeaponRect() { ... }
public void MmgDraw(MmgPen p) { ... }

//Support Methods
public MmgBmp GetSubjFront() { ... }
public void SetSubjFront(MmgBmp SubjFront) { ... }
public MmgBmp GetSubjBack() { ... }
public void SetSubjBack(MmgBmp SubjBack) { ... }
```

```
public MmgBmp GetSubjLeft() { ... }
public void SetSubjLeft(MmgBmp SubjLeft) { ... }
public MmgBmp GetSubjRight() { ... }
public void SetSubjRight(MmgBmp SubjRight) { ... }
public MdtChar GetHolder() { ... }
public void SetHolder(MdtChar Holder) { ... }
public boolean GetIsActive() { ... }
public void SetIsActive(boolean b) { ... }

public MdtWeaponType GetWeaponType() { ... }
public void SetWeaponType(MdtWeaponType WeaponType) { ... }

public long GetAnimTimeMsCurrent() { ... }
public void SetAnimTimeMsCurrent(long h) { ... }
public long GetAnimTimeMsTotal() { ... }
public void SetAnimTimeMsTotal(long h) { ... }
public double GetAnimPrctComplete() { ... }
public void SetAnimPrctComplete(double d) { ... }
public int GetDamage() { ... }
public void SetDamage(int d) { ... }
public MmgRect GetSrc() { ... }
public void SetSrc(MmgRect Src) { ... }
public MmgRect GetDst() { ... }
public void SetDst(MmgRect Dst) { ... }

public int GetThrowingSpeed() { ... }
public void SetThrowingSpeed(int h) { ... }
public int GetThrowingSpeedSkew() { ... }
public void SetThrowingSpeedSkew(int h) { ... }
public int GetThrowingFrame() { ... }
public void SetThrowingFrame(int h) { ... }
public int GetThrowingDir() { ... }
public void SetThrowingDir(int h) { ... }
public long GetThrowingCoolDown() { ... }
public void SetThrowingCoolDown(long h) { ... }
public long GetThrowingTimeMsCurrent() { ... }
```

```
public void SetThrowingTimeMsCurrent(long h) { ... }
public long GetThrowingTimeMsRotation() { ... }
public void SetThrowingTimeMsRotation(long h) { ... }

public long GetStabbingCoolDown() { ... }
public void SetStabbingCoolDown(long h) { ... }
public ScreenGame GetScreen() { ... }
public void SetScreen(ScreenGame Screen) { ... }
public MmgRect GetCurrent() { ... }
public void SetCurrent(MmgRect Current) { ... }
public MdtPlayerType GetPlayer() { ... }
public void SetPlayer(MdtPlayerType p) { ... }
public MdtWeaponAttackType GetAttackType() { ... }
public void SetAttackType(MdtWeaponAttackType AttackType) { ... }

public MdtWeaponPathType GetThrowingPath() { ... }
public void SetThrowingPath(MdtWeaponPathType ThrowingPath) { ... }

public Random GetRand() { ... }
public void SetRand(Random r) { ... }
```

The class definitions for both versions of the game engine are listed in the following. Be sure to check your `import/using` statements against the completed class project. If you're following along in C#, make sure to double-check your language syntax. Lastly, don't forget we still have to add a `ScreenGame` class to the project. So, if you have a lot of unresolved errors because of the missing class, just hang in there. We'll get to it.

Listing 22-41. MdtWeapon Class Definitions 1

```
//Java Version
public class MdtWeapon extends MdtBase {

//C# Version
public class MdtWeapon : MdtBase {
```

Now that that's out of the way, we'll quickly cover the class' support methods before we get into the main methods.

MdtWeapon: Support Method Details

The MdtWeapon class' support methods are simple get and set method pairs associated with class fields, and there are a lot of them. I'll list the support methods here, but we won't go into any detail reviewing them. Carefully add these methods to your copy of the MdtWeapon class or copy and paste them in from the completed MdtWeapon class included with this chapter's completed project. The project folder is part of the game engine's project folder. Let's look at some code.

Listing 22-42. MdtWeapon Support Method Details 1

```
01 public MmgBmp GetSubjFront() {
02     return subjFront;
03 }

01 public void SetSubjFront(MmgBmp SubjFront) {
02     subjFront = SubjFront;
03 }

01 public MmgBmp GetSubjBack() {
02     return subjBack;
03 }

01 public void SetSubjBack(MmgBmp SubjBack) {
02     subjBack = SubjBack;
03 }

01 public MmgBmp GetSubjLeft() {
02     return subjLeft;
03 }

01 public void SetSubjLeft(MmgBmp SubjLeft) {
02     subjLeft = SubjLeft;
03 }

01 public MmgBmp GetSubjRight() {
02     return subjRight;
03 }
```

```
01 public void SetSubjRight(MmgBmp SubjRight) {
02     subjRight = SubjRight;
03 }

01 public MdtChar GetHolder() {
02     return holder;
03 }

01 public void SetHolder(MdtChar Holder) {
02     holder = Holder;
03 }

01 public boolean GetIsActive() {
02     return active;
03 }

01 public void SetIsActive(boolean b) {
02     active = b;
03 }

01 public MdtWeaponType GetWeaponType() {
02     return weaponType;
03 }

01 public void SetWeaponType(MdtWeaponType WeaponType) {
02     weaponType = WeaponType;
03 }

01 public long GetAnimTimeMsCurrent() {
02     return animTimeMsCurrent;
03 }

01 public void SetAnimTimeMsCurrent(long h) {
02     animTimeMsCurrent = h;
03 }

01 public long GetAnimTimeMsTotal() {
02     return animTimeMsTotal;
03 }
```

```
01 public void SetAnimTimeMsTotal(long h) {
02     animTimeMsTotal = h;
03 }

01 public double GetAnimPrctComplete() {
02     return animPrctComplete;
03 }

01 public void SetAnimPrctComplete(double d) {
02     animPrctComplete = d;
03 }

01 public int GetDamage() {
02     return damage;
03 }

01 public void SetDamage(int d) {
02     damage = d;
03 }

01 public MmgRect GetSrc() {
02     return src;
03 }

01 public void SetSrc(MmgRect Src) {
02     src = Src;
03 }

01 public MmgRect GetDst() {
02     return dst;
03 }

01 public void SetDst(MmgRect Dst) {
02     dst = Dst;
03 }

01 public int GetThrowingSpeed() {
02     return throwingSpeed;
03 }
```

```
01 public void SetThrowingSpeed(int h) {
02     throwingSpeed = h;
03 }

01 public int GetThrowingSpeedSkew() {
02     return throwingSpeedSkew;
03 }

01 public void SetThrowingSpeedSkew(int h) {
02     throwingSpeedSkew = h;
03 }

01 public int GetThrowingFrame() {
02     return throwingFrame;
03 }

01 public void SetThrowingFrame(int h) {
02     throwingFrame = h;
03 }

01 public int GetThrowingDir() {
02     return throwingDir;
03 }

01 public void SetThrowingDir(int h) {
02     throwingDir = h;
03 }

01 public long GetThrowingCoolDown() {
02     return throwingCoolDown;
03 }

01 public void SetThrowingCoolDown(long h) {
02     throwingCoolDown = h;
03 }

01 public long GetThrowingTimeMsCurrent() {
02     return throwingTimeMsCurrent;
03 }
```

```
01 public void SetThrowingTimeMsCurrent(long h) {
02     throwingTimeMsCurrent = h;
03 }

01 public long GetThrowingTimeMsRotation() {
02     return throwingTimeMsRotation;
03 }

01 public void SetThrowingTimeMsRotation(long h) {
02     throwingTimeMsRotation = h;
03 }

01 public long GetStabbingCoolDown() {
02     return stabbingCoolDown;
03 }

01 public void SetStabbingCoolDown(long h) {
02     stabbingCoolDown = h;
03 }

01 public ScreenGame GetScreen() {
02     return screen;
03 }

01 public void SetScreen(ScreenGame Screen) {
02     screen = Screen;
03 }

01 public MmgRect GetCurrent() {
02     return current;
03 }

01 public void SetCurrent(MmgRect Current) {
02     current = Current;
03 }

01 public MdtPlayerType GetPlayer() {
02     return player;
03 }
```

```
01 public void SetPlayer(MdtPlayerType p) {
02     player = p;
03 }

01 public MdtWeaponAttackType GetAttackType() {
02     return attackType;
03 }

01 public void SetAttackType(MdtWeaponAttackType AttackType) {
02     attackType = AttackType;
03 }

01 public MdtWeaponPathType GetThrowingPath() {
02     return throwingPath;
03 }

01 public void SetThrowingPath(MdtWeaponPathType ThrowingPath) {
02     throwingPath = ThrowingPath;
03 }

01 public Random GetRand() {
02     return rand;
03 }

01 public void SetRand(Random r) {
02     rand = r;
03 }
```

There are a lot of support methods to review here. Don't feel like you have to type them up right now. I'll show you a shortcut where you can copy and paste the completed project file into your game project and save yourself some time. That brings us to the end of this review section. In the next section, we'll look at the `MdtWeapon` class' main methods.

MdtWeapon: Main Method Details

The `MdtWeapon` class has some constructors and other main methods we need to review. Let's take a look at the class constructors first.

Listing 22-43. MdtWeapon Main Method Details 1

```
01 public MdtWeapon(MdtChar Holder, MdtWeaponType WeaponType, MdtPlayerType
   Player) {
02     super();
03     SetMdtType(MdtObjType.WEAPON);
04     SetPlayer(Player);
05     SetHolder(Holder);
06     SetWeaponType(WeaponType);
07     SetAttackType(MdtWeaponAttackType.STABBING);
08     SetRand(new Random(System.currentTimeMillis()));
09 }
```

```
01 public MdtWeapon(MdtChar Holder, MdtWeaponType WeaponType,
   MdtWeaponAttackType AttackType, MdtPlayerType Player) {
02     super();
03     SetMdtType(MdtObjType.WEAPON);
04     SetPlayer(Player);
05     SetHolder(Holder);
06     SetWeaponType(WeaponType);
07     SetAttackType(AttackType);
08     SetRand(new Random(System.currentTimeMillis()));
09 }
```

Just a quick note to those of you following along in C#: Pay close attention to the use of the Java `super` keyword in constructors like those listed in the preceding. In C#, these must be expressed as `base` class constructor calls as shown in the C# version of this chapter's completed project.

Also note that the syntax for the current time in milliseconds is different in C# where the `DateTimeOffset.UtcNow. ToUnixTimeMilliseconds` method is used for a similar time measurement. Other than the notes previously mentioned, these constructors are direct. They configure class field values based on the arguments passed to them. Let's move on to the next set of main methods for us to review.

Listing 22-44. MdtWeapon Main Method Details 2

```
@Override
01 public MdtWeapon Clone() {
02     MdtWeapon ret = new MdtWeapon(holder, weaponType, player);
03     ret.SetAnimPrctComplete(GetAnimPrctComplete());
04     ret.SetIsActive(GetIsActive());
05     ret.SetAnimTimeMsCurrent(GetAnimTimeMsCurrent());
06     ret.SetAnimTimeMsTotal(GetAnimTimeMsTotal());
07     ret.SetAttackType(GetAttackType());
08
09     if(GetMmgColor() == null) {
10         ret.SetMmgColor(GetMmgColor());
11     } else {
12         ret.SetMmgColor(GetMmgColor().Clone());
13     }
14
15     if(GetCurrent() == null) {
16         ret.SetCurrent(GetCurrent());
17     } else {
18         ret.SetCurrent(GetCurrent().Clone());
19     }
20
21     ret.SetDamage(GetDamage());
22     ret.SetHeight(GetHeight());
23     ret.SetHasParent(GetHasParent());
24     ret.SetIsVisible(GetIsVisible());
25     ret.SetId(GetId());
26     ret.SetHolder(GetHolder());
27     ret.SetParent(GetParent());
28
29     if(GetPosition() == null) {
30         ret.SetPosition(GetPosition());
31     } else {
```

```
32          ret.SetPosition(GetPosition().Clone());
33      }
34
35      if(GetSubjBack() == null) {
36          ret.SetSubjBack(GetSubjBack());
37      } else {
38          ret.SetSubjBack(GetSubjBack().CloneTyped());
39      }
40
41      if(GetSubjFront() == null) {
42          ret.SetSubjFront(GetSubjFront());
43      } else {
44          ret.SetSubjFront(GetSubjFront().CloneTyped());
45      }
46
47      if(GetSubjLeft() == null) {
48          ret.SetSubjLeft(GetSubjLeft());
49      } else {
50          ret.SetSubjLeft(GetSubjLeft().CloneTyped());
51      }
52
53      if(GetSubjRight() == null) {
54          ret.SetSubjRight(GetSubjRight());
55      } else {
56          ret.SetSubjRight(GetSubjRight().CloneTyped());
57      }
58
59      ret.SetThrowingDir(GetThrowingDir());
60      ret.SetThrowingFrame(GetThrowingFrame());
61      ret.SetThrowingPath(GetThrowingPath());
62      ret.SetThrowingSpeed(GetThrowingSpeed());
63      ret.SetThrowingSpeedSkew(GetThrowingSpeedSkew());
64      ret.SetThrowingCoolDown(GetThrowingCoolDown());
65      ret.SetThrowingTimeMsRotation(GetThrowingTimeMsRotation());
```

```
66      ret.SetThrowingTimeMsCurrent(GetThrowingTimeMsCurrent());
67      ret.SetScreen(GetScreen());
68      ret.SetStabbingCoolDown(GetStabbingCoolDown());
69      return ret;
70  }

@Override
001 public boolean MmgUpdate(int updateTick, long currentTimeMs, long
    msSinceLastFrame) {
002     lret = false;
003     if (isVisible == true && active == true) {
004         animTimeMsCurrent += msSinceLastFrame;
005
006         if(attackType == MdtWeaponAttackType.THROWING) {
007             if(current == null) {
008                 current = new MmgRect(holder.GetX() + holder.
                    GetWidth()/2 - GetWidth()/2, holder.GetY() + holder.
                    GetHeight()/2 - GetHeight()/2, holder.GetY() + holder.
                    GetHeight()/2 + GetHeight()/2, holder.GetX() + holder.
                    GetWidth()/2 + GetWidth()/2);
009                 throwingFrame = 0;
010             }
011
012             if(throwingSpeed < 0) {
013                 throwingSpeed *= -1;
014             }
015
016             throwingTimeMsCurrent += msSinceLastFrame;
017
018             if(throwingPath == MdtWeaponPathType.NONE) {
019                 tmpI = rand.nextInt(11);
020                 throwingDir = holder.dir;
021                 if(tmpI % 2 == 0) {
022                     throwingPath = MdtWeaponPathType.PATH_1;
023                 } else if(tmpI % 3 == 0) {
024                     throwingPath = MdtWeaponPathType.PATH_2;
```

```
025                 } else {
026                     throwingPath = MdtWeaponPathType.PATH_3;
027                 }
028             } else {
029                 if(throwingDir == MmgDir.DIR_BACK) {
030                     if(throwingPath == MdtWeaponPathType.PATH_1) {
031                         current.ShiftRect((throwingSpeedSkew * -1),
                            (throwingSpeed * -1));
032                     } else if(throwingPath == MdtWeaponPathType.PATH_2) {
033                         current.ShiftRect(0, (throwingSpeed * -1));
034                     } else if(throwingPath == MdtWeaponPathType.PATH_3) {
035                         current.ShiftRect((throwingSpeedSkew * 1),
                            (throwingSpeed * -1));
036                     }
037                 } else if(throwingDir == MmgDir.DIR_FRONT) {
038                     if(throwingPath == MdtWeaponPathType.PATH_1) {
039                         current.ShiftRect((throwingSpeedSkew * -1),
                            (throwingSpeed * 1));
040                     } else if(throwingPath == MdtWeaponPathType.PATH_2) {
041                         current.ShiftRect(0, (throwingSpeed * 1));
042                     } else if(throwingPath == MdtWeaponPathType.PATH_3) {
043                         current.ShiftRect((throwingSpeedSkew * 1),
                            (throwingSpeed * 1));
044                     }
045                 } else if(throwingDir == MmgDir.DIR_LEFT) {
046                     if(throwingPath == MdtWeaponPathType.PATH_1) {
047                         current.ShiftRect((throwingSpeed * -1),
                            (throwingSpeedSkew * 1));
048                     } else if(throwingPath == MdtWeaponPathType.PATH_2) {
049                         current.ShiftRect((throwingSpeed * -1), 0);
050                     } else if(throwingPath == MdtWeaponPathType.PATH_3) {
051                         current.ShiftRect((throwingSpeed * -1),
                            (throwingSpeedSkew * -1));
052                     }
```

```
053                 } else if(throwingDir == MmgDir.DIR_RIGHT) {
054                     if(throwingPath == MdtWeaponPathType.PATH_1) {
055                         current.ShiftRect((throwingSpeed * 1),
                            (throwingSpeedSkew * 1));
056                     } else if(throwingPath == MdtWeaponPathType.PATH_2) {
057                         current.ShiftRect((throwingSpeed * 1), 0);
058                     } else if(throwingPath == MdtWeaponPathType.PATH_3) {
059                         current.ShiftRect((throwingSpeed * 1),
                            (throwingSpeedSkew * -1));
060                     }
061                 }
062
063                 if(animTimeMsCurrent > throwingTimeMsRotation) {
064                     animTimeMsCurrent = 0;
065                     throwingFrame++;
066                     if(throwingFrame > 3) {
067                         throwingFrame = 0;
068                     }
069                 }
070
071                 SetPosition(current.GetLeft(), current.GetTop());
072                 coll = screen.CanMove(current, this);
073                 if(coll != null) {
074                     if(coll.GetMdtType() == MdtObjType.ENEMY) {
075                         if(screen != null) {
076                             screen.UpdateProcessWeaponCollision(coll,
                                this, current);
077                             screen.UpdateRemoveObj(this, GetPlayer());
078                         }
079                     }
080                 }
081
082                 if(
083                     GetX() < ScreenGame.BOARD_LEFT
084                     || GetX() + GetWidth() > ScreenGame.BOARD_RIGHT
```

```
085                     || GetY() < ScreenGame.BOARD_TOP
086                     || GetY() + GetHeight() > ScreenGame.BOARD_BOTTOM
087                 ) {
088                     if(screen != null) {
089                         screen.UpdateRemoveObj(this, GetPlayer());
090                     }
091                 }
092             }
093         }
094
095         animPrctComplete = (double)animTimeMsCurrent / (double)
            animTimeMsTotal;
096         if(animPrctComplete > 1.0d) {
097             animPrctComplete = 1.0d;
098         }
099
100         lret = true;
101     }
102     return lret;
103 }
```

```
01 public MmgRect GetWeaponRect() {
02     if(attackType == MdtWeaponAttackType.STABBING) {
03         if(holder.dir == MmgDir.DIR_BACK) {
04             src = new MmgRect(0, 0, (int)(subjBack.GetHeight() *
                animPrctComplete), subjBack.GetWidth());
05             dst = new MmgRect((holder.GetX() + MmgHelper.ScaleValue(8) +
                holder.GetWidth()/2 - src.GetWidth()/2)
06                     , (holder.GetY() + MmgHelper.ScaleValue(8) - src.
                       GetHeight())
07                     , (holder.GetY() + MmgHelper.ScaleValue(8))
08                     , (holder.GetX() + MmgHelper.ScaleValue(8) + holder.
                       GetWidth()/2 + src.GetWidth()/2)
09                     );
10         } else if(holder.dir == MmgDir.DIR_FRONT) {
```

```
11              src = new MmgRect(0, subjFront.GetHeight() - (int)
                (subjFront.GetHeight() * animPrctComplete), subjFront.
                GetHeight(), subjFront.GetWidth());
12              dst = new MmgRect((holder.GetX() - MmgHelper.ScaleValue(8) +
                holder.GetWidth()/2 - src.GetWidth()/2)
13                      , (holder.GetY() - MmgHelper.ScaleValue(12) +
                        holder.GetHeight())
14                      , (holder.GetY() - MmgHelper.ScaleValue(12) +
                        holder.GetHeight() + src.GetHeight())
15                      , (holder.GetX() - MmgHelper.ScaleValue(8) + holder.
                        GetWidth()/2 + src.GetWidth()/2)
16                      );
17          } else if(holder.dir == MmgDir.DIR_LEFT) {
18              src = new MmgRect(0, 0, subjLeft.GetHeight(), (int)
                (subjLeft.GetWidth() * animPrctComplete));
19              dst = new MmgRect((holder.GetX() + MmgHelper.ScaleValue(8) -
                src.GetWidth())
20                      , (holder.GetY() + MmgHelper.ScaleValue(8) + holder.
                        GetHeight()/2 - src.GetHeight()/2)
21                      , (holder.GetY() + MmgHelper.ScaleValue(8) + holder.
                        GetHeight()/2 + src.GetHeight()/2)
22                      , (holder.GetX() + MmgHelper.ScaleValue(8))
23                      );
24          } else if(holder.dir == MmgDir.DIR_RIGHT) {
25              src = new MmgRect(subjRight.GetWidth() - (int)(subjRight.
                GetWidth() * animPrctComplete), 0, subjRight.GetHeight(),
                subjRight.GetWidth());
26              dst = new MmgRect((holder.GetX() + holder.GetWidth() -
                MmgHelper.ScaleValue(8))
27                      , (holder.GetY() + MmgHelper.ScaleValue(8) + holder.
                        GetHeight()/2 - src.GetHeight()/2)
28                      , (holder.GetY() + MmgHelper.ScaleValue(8) + holder.
                        GetHeight()/2 + src.GetHeight()/2)
29                      , (holder.GetX() + holder.GetWidth() - MmgHelper.
                        ScaleValue(8) + src.GetWidth())
```

```
30                      );
31          }
32      } else if(attackType == MdtWeaponAttackType.THROWING && weaponType
        == MdtWeaponType.AXE) {
33          if(throwingFrame == 0) {
34              dst = new MmgRect(subjBack.GetX(), subjBack.GetY(),
                subjBack.GetY() + subjBack.GetHeight(), subjBack.GetX() +
                subjBack.GetWidth());
35          } else if(throwingFrame == 1) {
36              dst = new MmgRect(subjFront.GetX(), subjFront.GetY(),
                subjFront.GetY() + subjFront.GetHeight(), subjFront.GetX() +
                subjFront.GetWidth());
37          } else if(throwingFrame == 2) {
38              dst = new MmgRect(subjLeft.GetX(), subjLeft.GetY(),
                subjLeft.GetY() + subjLeft.GetHeight(), subjLeft.GetX() +
                subjLeft.GetWidth());
39          } else if(throwingFrame == 3) {
40              dst = new MmgRect(subjRight.GetX(), subjRight.GetY(),
                subjRight.GetY() + subjRight.GetHeight(), subjRight.GetX() +
                subjRight.GetWidth());
41          }
42      }
43      return dst;
44  }
```

```
@Override
01 public void MmgDraw(MmgPen p) {
02      if (isVisible == true && active == true) {
03          GetWeaponRect();
04          if(attackType == MdtWeaponAttackType.STABBING) {
05              if(holder.dir == MmgDir.DIR_BACK) {
06                  p.DrawBmp(subjBack, src, dst);
07              } else if(holder.dir == MmgDir.DIR_FRONT) {
08                  p.DrawBmp(subjFront, src, dst);
09              } else if(holder.dir == MmgDir.DIR_LEFT) {
10                  p.DrawBmp(subjLeft, src, dst);
```

```
11              } else if(holder.dir == MmgDir.DIR_RIGHT) {
12                  p.DrawBmp(subjRight, src, dst);
13              }
14          } else if(attackType == MdtWeaponAttackType.THROWING &&
            weaponType == MdtWeaponType.AXE) {
15              if(throwingFrame == 0) {
16                  p.DrawBmp(subjBack);
17              } else if(throwingFrame == 1) {
18                  p.DrawBmp(subjFront);
19              } else if(throwingFrame == 2) {
20                  p.DrawBmp(subjLeft);
21              } else if(throwingFrame == 3) {
22                  p.DrawBmp(subjRight);
23              }
24          }
25      }
26  }
```

The main methods in the block listed in the preceding are fairly complex and long in length. I'll reiterate that you can copy and paste this class from the completed Chapter 22 project included in the game engine's main project folder. Just make sure that you read over and understand the code before doing so. The first main method for us to review in this block is the Clone method.

This implementation is very similar to the cloning methods we've seen in Part 1 of this text. The method uses the current object to create a new, unique version of the `MdtWeapon` class with the same class field values. This method is used by the DungeonTrap game to duplicate the throwing weapon's projectile so that the original weapon is not lost.

The next main method for us to cover is the `MmgUpdate` method. This method is used to update the weapon's attack animation. There are a few caveats regarding the `MdtWeapon` class that I'd like to discuss. The class is designed to work with four images. It generates the weapon animations using only those four subject images. When stabbing, the image is extended from the holder using the percentage animation complete to calculate where the weapon should be drawn.

When stabbing, the weapon is never actually repositioned; it is simply drawn based on the position of the holder and the progress of the stabbing animation. When the weapon is configured as a throwing weapon, then the projectile is animated as rotating end over end, and the weapon is actually repositioned as it travels across the screen.

On line 4 of the `MmgUpdate` method, we can see that the animation timing is running if the weapon is visible and active. If the type of attack is a throwing attack, then we prep the current rectangle, which represents a box around the weapon. The throwing frame is set to zero on line 9. This just sets the throwing animation to the first frame of four frames. There is one animation frame per cardinal direction.

The code on lines 12–14 ensures that the throwing speed is positive, and on line 16, the time duration of the current throw is updated. Note that fields are initialized if they are null, and on line 18 if no throwing path has been set, a random path is determined on lines 19–27.

The large block of code on lines 29–61 moves the weapon projectile in the thrown direction and on the thrown path. If the code seems too complex, try and view it one direction at a time, and the pattern to the code should become clear. If you direct your attention to lines 63–69, the animation for rotating the projectile is performed. The last few lines of code ensure that the projectile doesn't fly off of the game board.

The weapon detects collisions with enemy targets on lines 72–80. Notice that a centralized method of the `screen` class field is used to determine collisions and process weapon strikes. This is because these actions require a central class that is aware of all the game objects. The `ScreenGame` class is such a class. The last thing to discuss with regard to this method is the code on lines 95–98. This is all the code needed to process the stabbing attack animation. Keep this in mind when we look at the next two methods.

The next method up for review is the `GetWeaponRect` method. If a stabbing attack is underway, then the position of the weapon is drawn point first as emerging from the holding character. All the positioning in the code in this part of the method is based on the holder and the animation's progress over time. If the weapon is thrown however, the destination rectangle, where the weapon will be drawn, is based on the position of the `MdtWeapon` class itself.

Can you see why we don't have a set of methods that synchronize the position of the `MdtWeapon` class with all of its subject images? The short answer is because it just doesn't make sense in this case. The weapons all face in different directions, and keeping their position synchronized isn't useful as a stabbing weapon. The next main method for us to look at is the `MmgDraw` method.

For stabbing attacks, the animation is inherent in the drawing calls on lines 5–13. Notice that a call to the `GetWeaponRect` method is made at the start of the method. This ensures that the `dst` rectangle is prepped and ready for the next game frame. For a given direction, the weapon is drawn extending from the holder, end first, with a percentage visible that matches the animation percentage calculation. This means that at 100% the weapon is fully extended from the holder.

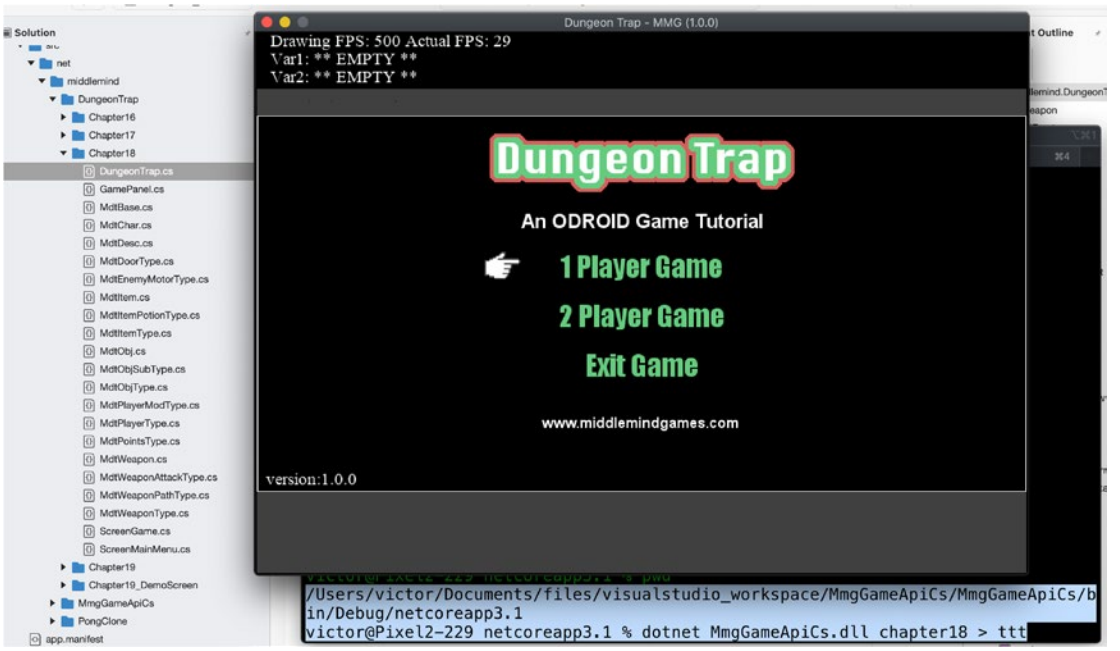
In the case of a throwing attack, the subject image associated with the current throwing frame is drawn. This is also an inherent animation. The image changes over time as the `throwingFrame` class field is incremented, which also happens over time. That brings us to the conclusion of the `MdtWeapon` class' review. It also marks the end of the base class review.

To quickly grab a copy of the code reviewed here and add it to your project, open the folder that contains the game engine project. Find the "cfg" folder and locate the "asset_src" folder inside. Open it. Now find and open the "dungeon_trap_base_classes" folder. Copy all the files inside and paste them into your project alongside the existing class files. To get a copy of the `ScreenGame` class with placeholder methods and fields stubbed out, just look in the DungeonTrap Chapter 21 folder and find the `ScreenGame` class. Copy and paste it into your project. Make sure the `package` or `namespace` of the newly copied files is adjusted to match that which your project uses.

You should now be able to compile your project with 23 files, 22 if you're using C#, and watch it load up. It won't look much different than the project results at the end of Chapter 21, but there's a lot going on under the hood. If you hang in there, in Chapter 23, we'll finish adding in the level 1 and 2 extended classes and demo the new game functionality. I've taken a screenshot of the current project running with all the new classes we've added to the project in the background.

this figure will be printed in b/w

The following screenshot shows the C# version of the project running the Chapter 22 code with all the new classes listed in the background.



this figure will be printed in b/w

We haven't crossed a lot of game specifications off the list in this chapter, but we're getting there. We've added a ton of functionality in the form of base classes. Don't worry. We'll soon have some fully functioning classes to demo!